Operating System/2™

Version 1.3

Procedures Language 2/REXX Reference

Programming Family

Operating System/2™

Version 1.3

Procedures Language 2/REXX Reference

Programming Family

First Edition (September 1990)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

© Copyright International Business Machines Corporation 1983, 1990. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Special Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

Operating System/2	OS/2
Operating System/400	OS/400
Systems Application Architecture	SAA
Enterprise Systems Architecture/370	PS/2

© Copyright IBM Corp. 1983, 1990

Contents

© Copyright IBM Corp. 1983, 1990

Chapter 1. Introduction

This introductory section:

- Identifies the book's purpose and audience
- Gives a brief overview of the Systems Application Architecture* (SAA)* specification
- Explains how to use the book.

Who Should Read This Book

The book is designed for experienced programmers, particularly those who have used a block-structured, high-level language (for example, PL/I, Algol, C, or Pascal).

This book describes the OS/2* Procedures Language processor and the REstructured eXtended eXecutor (abbreviated REXX) language. Descriptions include the use and syntax of the language and explain how the language processor "interprets" the language as a program is executing.

For ease of reference, the material in this book is arranged in chapters:

1. Introduction
2. General Concepts
3. Keyword Instructions (in alphabetical order)
4. Functions (in alphabetical order)
5. Parsing (a method of dividing character strings, such as commands)
6. Numerics and Arithmetic
7. Conditions and Condition Traps
8. Input and Output Streams
9. Applications Programming Interface
10. Debug Aids
11. Reserved Keywords and Special Variables
12. Some Useful OS/2 Commands.

There are also appendixes covering:

- Error Numbers and Messages
- Double-Byte Character Set (DBCS).

Using Systems Application Architecture

Systems Application Architecture is an IBM specification that defines a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications with cross-system consistency. The SAA Procedures Language has been defined as a subset of the REXX language that can be used in a number of computing environments.

If you are using REXX only in an OS/2 environment, this subset has no effect on your programs. If you plan on running your programs in other environments, however, some restrictions may apply. We suggest that you consult the *SAA Common Programming Interface Procedures Language Reference.*

© Copyright IBM Corp. 1983, 1990

The Systems Application Architecture specification:

- Defines a **common programming interface** you can use to develop applications that can be integrated with each other and transported to run in multiple SAA environments.
- Defines **common communications support** that you can use to connect applications, systems, networks, and devices.
- Defines a **common user access** that you can use to achieve consistency in panel layout and user interaction techniques.
- Offers some **common applications** written by IBM using the common programming interface, the common communications support, and the common user access.

Supported Environments

The SAA specification provides a framework across these IBM computing environments:

- MVS for TSO/E in the Enterprise Systems Architecture/370*
- CMS in the VM/System Product or VM/Extended Architecture
- Operating System/400* (OS/400*)
- Operating System/2* (OS/2) Extended Edition.

Common Programming Interface

As its name implies, the Common Programming Interface (CPI) provides languages, commands, and calls that programmers can use to develop applications that take advantage of the consistency the SAA specification offers. These applications can easily be integrated and transported across the supported environments.

The components of the interface fall into two general categories:

- Languages

  Application Generator
  C
  COBOL
  FORTRAN
  PL/I
  Procedures Language
  RPG.

- Services

  Communications Interface
  Database Interface
  Dialog Interface
  Presentation Interface
  Query Interface.

The CPI is not in itself a product or a piece of code. But—as a definition—it establishes and controls how IBM products are being implemented, and it establishes a common base across the SAA environments.

Thus, when you want to create an application that can be used in more than one environment, you can stay within the boundaries of the CPI and obtain easier portability. (Naturally, you design such applications with portability in mind as well.) In addition to the CPI, you may also want to consider the other aspects of the Systems Application Architecture specification—for example, the common user access—when creating your applications.

How to Read the Syntax Diagrams in This Book

Throughout this book, syntax is described using the structure as defined in the following text.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►— symbol indicates the beginning of a statement.

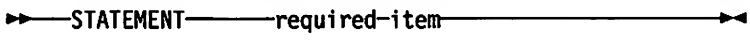
  The —► symbol indicates that the statement syntax continues.

  The ►— symbol indicates that a line continues from the previous line.

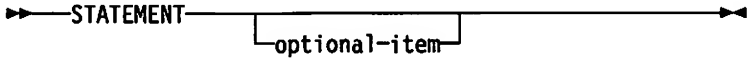
  The —►◄ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►— symbol and end with the —► symbol.

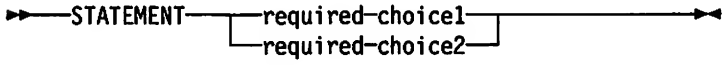
- Required items are shown on the horizontal line (the main path).



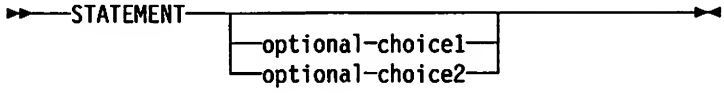
- Optional items are shown below the main path.



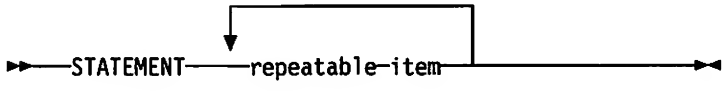
- When you can choose from two or more items, they are stacked vertically. If you must choose one of the items, an item of the stack is shown on the main path.



  If choosing one of the items is optional, the entire stack is shown below the main path.



- An arrow returning to the left above the main line indicates that you can repeat an item.



  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

- Keywords are shown in uppercase (for example, PARM1). They must be spelled exactly as shown. Variables are shown in all lowercase letters (for example, parmx). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, they must be entered as part of the syntax.

For More REXX Information

Here is a list of books that you may wish to include in your REXX library:

- The *SAA Common Programming Interface Procedures Language Reference* can be useful to more experienced REXX users who may wish to code portable programs. This book defines the SAA Procedures Language. Descriptions include the use and syntax of the language as well as explanations on how the language processor interprets the language as a program is executing.
- The *TSO/E Version 2 REXX Reference* is a comprehensive reference for use on TSO/E.
- The *VM/SP System Product Interpreter Reference* is a comprehensive reference for use with the System Product Interpreter on VM/SP.
- The *OS/2 Procedures Language 2/REXX User's Guide* offers a general introduction to programming for beginners and extensive practical examples of REXX applications for OS/2 programmers of all levels.

Chapter 2. General Concepts

The REstructured eXtended eXecutor (REXX) language is a language particularly suitable for:

- Command procedures
- Application front ends
- User-defined macros (such as Dialog Manager and editor subcommands)
- Prototyping
- Personal computing.

It is a general purpose programming language similar to PL/I. REXX has the usual structured programming instructions—IF, SELECT, DO WHILE, LEAVE, and so on—and a number of useful built-in functions.

No restrictions are imposed by the language on program format. There can be more than one clause on a line or a single clause can occupy more than one line. Indentation is allowed. Programs can, therefore, be coded in a format that emphasizes their structure, making them easier to read.

There is no limit to the length of the values of variables, as long as all variables fit into the storage available. Symbols (variable names) are limited to a length of 250 characters.

Compound symbols, such as `NAME.X.Y` (where X and Y can be the names of variables), can be used for constructing arrays and for other purposes.

REXX programs are executed by a language processor (interpreter). That is, the program is executed line-by-line and word-by-word, without first being translated to another form (compiled). The advantage of this to the user is that if the program fails with a syntax error of some kind, the point of failure is clearly indicated; usually, it does not take long to understand the problem and make a correction.

REXX and the OS/2 Program

REXX has been designed to be an integral part of the OS/2 program. There is no installation process or explicit invocation of the language processor. REXX program files have the default extension *CMD*, and they can contain OS/2 commands as well as REXX instructions. Anywhere an OS/2 command or batch-file is used, a REXX program can be used.

Structure and General Syntax

Programs written in the REXX language must start with a comment in the first column of the first line. This distinguishes a REXX program from an OS/2 batch file.

A REXX program is built from a series of *clauses* that are composed of:

- Zero or more blanks (which are ignored)
- A sequence of tokens (see "Tokens" on page 8)
- Zero or more blanks (again ignored)

© Copyright IBM Corp. 1983, 1990

- A semicolon (;) delimiter that may be implied by line-end, certain keywords, or the colon (:) if it follows a single symbol.

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to special characters, including operators (see page 10), are also removed.

Tokens

Programs written in REXX are composed of tokens (of any length, up to an implementation-restricted maximum) that are separated by blanks or by the nature of the tokens themselves. The classes of tokens are:

Comments

Comments are a sequence of characters (on one or more lines) delimited by /* and */. Comments can contain other comments, as long as each begins and ends with the necessary delimiters. You can write comments anywhere in a program. The language processor ignores them (and hence they can be of any length), but they do act as separators.

```
/* This is an example of a valid comment */
```

Literal Strings

A literal string is a sequence including any character except linefeed (X'10') and delimited by the single quote (') or the double quote ("). Use two consecutive double quotes ("") to represent a " character within a string delimited by double quotes. Similarly, use two consecutive single quotes ('') to represent a ' character within a string delimited by single quotes. A literal string is a constant and its contents are never modified when it is processed. Literal strings must be contained on a single line (this means that unmatched quotes can be detected on the line where they occur).

A literal string with no characters (that is, a string of length 0) is called a *null string*.

These are valid literal strings:

```
'Fred'
"Don't Panic!"
'You shouldn''t'        /* Same as "You shouldn't" */
```

Implementation maximum: A literal string can contain up to 250 characters. The length of the evaluated result of an expression, however, can be up to 4 billion bytes.

Hexadecimal Strings

Hexadecimal strings are any sequence of zero or more hexadecimal digits (0-9, a-f, A-F), separated by blanks, delimited by single or double quotes, and immediately followed by the symbol x or X (the X cannot be part of a longer symbol). A single leading 0 is added, if necessary, at the front of the string to make an even number of hexadecimal digits, which represent a character-string constant formed by packing the hexadecimal codes given. The blanks, which may be present only at byte boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

These are valid hexadecimal strings:

```
'ABCD'x
"1d ec f8"X
"1 d8"x
```

Note: If you use explicit hexadecimal strings, your program may operate differently when ported to different machines.

Implementation maximum: The packed length of a hexadecimal string cannot exceed 250 bytes.

Binary strings

Binary strings are any sequence of zero or more binary digits (0 or 1) in groups of 8 (bytes) or 4 (nibbles), optionally separated by one or more blanks. The entire string is delimited by matching single or double quotes and immediately followed by the symbol b or B (which cannot be part of a longer symbol). The blanks, which may only be present at byte or nibble boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

These are valid binary-literal strings:

```
'0010'b
"10 1101 0101"B
"01011011 10011010"b
```

Implementation maximum: The packed length of a binary-literal string may not exceed 100 bytes.

Symbols

Symbols are groups of characters, selected from the English alphabetic and numeric characters (A-Z, a-z, 0-9) and from the characters . ! ? and _(underscore). Any lowercase alphabetic character in a symbol is translated to uppercase (for example, a lowercase a-z becomes an uppercase A-Z).

These are valid symbols:

```
Fred
Albert.Hall
WHERE?
```

A symbol can be a label (see page 17) or a REXX keyword (see page 207). If a symbol does not begin with a digit or a period, you can use it as a variable and can assign it a value. If you have not assigned it a value, its value is the characters of the symbol itself, translated to uppercase (for example, a lowercase a-z becomes an uppercase A-Z). Symbols that begin with a number or a period are constant symbols and cannot be assigned a value. There is one other type of symbol. If the first part of a symbol starts with a digit (0-9) or a period, it may end with the sequence beginning with E (uppercase or lowercase), followed by an optional sign (− or +), followed by one or more digits (which cannot be followed by any other symbol characters). This type of symbol, subject to validation, is assumed to be a number in exponential notation. The sign in this context is part of the symbol and is not an operator.

These are valid exponential symbols:

```
17.3E-12
.03e+9
```

Numbers

Numbers are character strings consisting of one or more decimal digits optionally prefixed by a plus (+) or minus (−) sign and optionally including a single period (.) that represents a decimal point. A number can also have a power of 10 suffixed in conventional exponential notation: an E (uppercase or lowercase), followed optionally by a plus (+) or minus (−) sign, and then followed by one or more decimal digits defining the power of 10. Whenever a character string is used as a number, rounding may occur to a precision specified by the NUMERIC DIGITS instruction (the default is nine digits). See pages 125 through 134 for a full definition of numbers.

Numbers can have leading blanks (before and after the sign, if any) and can have trailing blanks. Embedded blanks are not permitted. Note that a symbol (see the preceding text) or a literal string can be a number. A number cannot be the name of a variable.

These are valid numbers:

```
12
-17.9
127.0650
73e+128
' + 7.9E5 '
```

A *whole number* is a number that has a zero (or no) decimal part and that the language processor would not normally express in exponential notation. That is, it has no more digits before the decimal point than the current setting of NUMERIC DIGITS (the default is 9).

Implementation maximum: The exponent of a number expressed in exponential notation can have up to nine digits.

Operators

Operators are special characters + − \ * / % * | ¦ & = ¬ > <. The sequences >= <= \> ¬> \< ¬< \= ¬= >< <> == \== ¬== // && ** >> << >>= \>> ¬>> <<= ¦¦ \<< ¬<< are operator tokens (see page 12), with or without embedded blanks or comments. One or more blanks, where they occur in expressions but are not adjacent to another operator, also act as an operator.

Some of these characters may not be available in all character sets, and, if this is the case, appropriate translations can be used. In particular, the vertical bar *or* symbol is often shown as a split vertical bar.

Note that throughout the language, the *not* symbol (¬) is synonymous with the backslash (\). The two symbols may be used interchangeably according to availability and personal preference.

Special Characters

Special characters are the characters , ; :) (together with the individual characters from the operators have special significance when found outside of strings. All these characters constitute the set of special characters. They all act as token delimiters and blanks adjacent to any of these are removed, with the exception that a blank adjacent to the outside of a parenthesis is deleted only if it is also adjacent to another special character (unless this is a parenthesis and the blank is outside of it, also).

For example, the clause:

```
'REPEAT'   B + 3;
```

is composed of six tokens—a string ('REPEAT'), a blank operator, a symbol (B, which may have a value), an operator (+), a second symbol (3, which is a number and a symbol), and the clause delimiter (;). The blanks between B and + and between + and 3 are removed. However, one of the blanks between REPEAT and B remains as an operator. Thus, this clause is treated as though it were written:

```
'REPEAT' B+3;
```

Implied Semicolons

The last element in a clause is the semicolon delimiter. The language processor implies the semicolon in three cases: by a line-end, after certain keywords, and after a colon if it follows a single symbol. This means that you need to include semicolons only when there is more than one clause on a line.

A line-end usually marks the end of a clause; thus, a semicolon is implied at most line-ends. However, there are exceptions:

- The line ends in the middle of a comment.
- The last noncomment token was the continuation character (denoted by a comma).

In these situations, a line-end is not considered the end of a clause and a semicolon is not implied.

Semicolons are also implied automatically after certain keywords when they are used in the correct context. The keywords that have this effect are ELSE, OTHERWISE, and THEN. These special cases reduce typographical errors significantly.

Note: The two characters forming the comment delimiters, /* and */, must not be split by a line-end (that is, / and * must not be displayed on different lines) since they could not then be recognized correctly: an implied semicolon would be added. The two characters forming a double quote within a string are also subject to this line-end ruling.

Continuations

One way to continue a clause onto the next line is to use the comma, which is referred to as the *continuation character*. The comma is functionally replaced by a blank; thus, no semicolon is implied.

The following example shows how to use the continuation character to continue a clause:

```
say 'You can use a comma',
    'to continue this clause.'
```

This causes the following to be displayed:

```
You can use a comma to continue this clause.
```

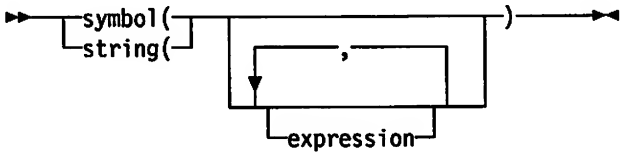
Expressions and Operators

Expressions

Clauses can include expressions consisting of *terms* (strings, symbols, and function calls) interspersed with operators and parentheses.

Terms include:

- **Literal strings** (delimited by quotes), which are literal constants.
- **Symbols** (no quotes), which are translated to uppercase. Those that do not begin with a digit or a period may be the name of a variable; in this case, the value of that variable replaces them as soon as they are needed during evaluation. Otherwise, they are treated as a literal string. A symbol can also be *compound.*
- **Function invocations** (see page 65), which are of the form:



Evaluation of an expression is left-to-right, modified by parentheses and by operator precedence in the usual algebraic manner (see "Parentheses and Operator Precedence" on page 15). Expressions are always completely evaluated, unless an error occurs during evaluation.

All data is in the form of *typeless* character strings (typeless because it is not—as in some other languages—of a particular declared type, such as Binary, Hexadecimal, Array, and so on). Consequently, the result of evaluating any expression is itself a character string. All terms and results may be the null string (a string of length 0). Note that REXX imposes no restriction on the maximum length of results, but there is usually some practical limitation dependent upon the amount of storage available to the language processor.

Operators

The following pages describe how each operator (except for the prefix operators) acts on two terms, which may be symbols, strings, function calls, intermediate results, or subexpressions in parentheses. Each prefix operator acts on the term or subexpression that follows it. There are four types of operators: string concatenation, arithmetic, comparison, and logical (Boolean).

String Concatenation

The concatenation operators combine two strings to form one string. The combination can occur with or without an intervening blank:

(blank) Concatenate terms with one blank in between

|| or ¦¦ Concatenate without an intervening blank

(abuttal) Concatenate without an intervening blank.

You can force concatenation without a blank by using the ¦¦ operator.

The *abuttal* operator is assumed between terms that are not separated by another operator. Any comments between the terms are irrelevant. For example:

If the variable FRED has the value 37.4, then Fred"%" evaluates to 37.4%.

If the variable PETER has the value 1, then (Fred)(Peter) evaluates to 37.41.

In ASCII, the two adjoining strings, one hexadecimal and one literal, '4a 4b'x'LMN' evaluate to 'JKLMN'.

In the case of:

```
Fred/* The NOT operator precedes Peter. */¬Peter
```

there is no abuttal operator implied; it is an invalid expression. However,

```
(Fred)/* The NOT operator precedes Peter. */(¬Peter)
```

results in an abuttal and evaluates to 37.40.

Arithmetic

You can combine character strings that are valid numbers (see page 10) using the arithmetic operators:

+	Add.
−	Subtract.
*	Multiply.
/	Divide.
%	Divide and return the integer part of the result.
//	Divide and return the remainder (not modulo, since the result may be negative).
**	Power (raise a number to a whole-number power).
Prefix −	Negate the following term. Same as 0−term.
Prefix +	Take following term as if it was 0+term.

See Chapter 6, "Numerics and Arithmetic," for details of accuracy, the format of valid numbers, and the combination rules for arithmetic operators. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

Comparison

The comparison operators return the value 1 if the result of the comparison is true, or 0 if otherwise.

The = =, \= =, and ¬ = = operators test for an exact match between two strings. In this case, the two strings must be identical before they are considered strictly equal. Similarly, the strict comparison operators such as > > or < < carry out a simple character-by-character comparison, with no padding of either of the strings being compared. The comparison of the two strings is from left to right. The strict comparison operators do not attempt to perform a numeric comparison on the two operands.

For all the other comparison operators, if *both* terms involved are numeric, a numeric comparison (in which leading zeros are ignored, and so on) is effected; otherwise, both terms are treated as character strings (leading and trailing blanks are ignored and then the shorter string is padded with blanks on the right). The character comparison operation is case-sensitive, and (as in strict comparisons) the exact collating order depends on the character set used for the implementation. For example, in an EBCDIC environment, lowercase alphabetics precede uppercase, and the digits 0-9 are higher than all alphabetics. In an ASCII environment, the digits are lower than the alphabetics and lowercase alphabetics are higher than uppercase alphabetics.

The comparison operators and operations are:

= =	True if terms are strictly equal (identical).
=	True if the terms are equal (numerically or when padded).
\= = or ¬ = =	True if the terms are *not* strictly equal (inverse of = =).
\= or ¬ =	Not equal (inverse of =).
>	Greater than.
<	Less than.
> >	Strictly greater than.
< <	Strictly less than.
> <	Greater than or less than (same as \= or ¬ =).
< >	Greater than or less than (same as \= or ¬ =).
> =	Greater than or equal to.
\< or ¬ <	Not less than.
> > =	Strictly greater than or equal to.
\< < or ¬ < <	Strictly *not* less than.
< =	Less than or equal to.
\> or ¬ >	Not greater than.
< < =	Strictly less than or equal to.
\> > or ¬ > >	Strictly *not* greater than.

Note: Throughout the language, the *not* symbol (¬) is synonymous with the backslash (\). You can use the two symbols interchangeably according to availability and personal preference. The backslash can be contained in the following operators: `\(prefix not)`, `\=`, `\==`, `\<`, `\>`, `\<<`, and `\>>`.

Logical (Boolean)

A character string is taken as *false* if it has a value of 0, and *true* if it has a value of 1. The logical operators take one or two such values (values other than 0 or 1 are not allowed) and return 0 or 1 as appropriate:

& AND
Returns 1 if both terms are true.

| or **¦** Inclusive-OR
Returns 1 if either term is true.

&& Exclusive-OR
Returns 1 if either (but not both) is true.

Prefix ** or **¬ Logical NOT
Negates; 1 becomes 0 or 0 becomes 1.

Note: On ASCII systems (for example, IBM PS/2* systems), REXX recognizes the ASCII character 124 as the logical OR symbol. Depending on the code page or keyboard you are using for your particular country, the logical OR symbol may be shown as a solid vertical bar (|) or a split vertical bar (¦). The appearance of the symbol on your screen may not match the symbol engraved on the key. If you are receiving error 13, `invalid character in program`, on an instruction including a vertical bar character, make sure this character is ASCII 124.

Parentheses and Operator Precedence

Expression evaluation is from left to right; parentheses and operator precedence modify this:

- When parentheses are encountered (other than those that identify function calls), the entire subexpression between the parentheses is evaluated immediately when the term is required.
- When the sequence

  `term1 operator1 term2 operator2 term3 ...`

  is encountered, and `operator2` has a higher precedence than `operator1`, the expression (`term2 operator2 term3 ...`) is evaluated first, applying the same rule repeatedly as necessary.

  Note, however, that individual *terms* are evaluated from left to right in the expression (that is, as soon as they are encountered). Only the order of *operations* is affected by the precedence rules.

For example, * (multiply) has a higher priority than + (add), so 3+2*5 evaluates to 13 (rather than the 25 that would result if strict left-to-right evaluation occurred). Likewise, the expression −3**2 evaluates to 9 (instead of -9) because the prefix minus operator has a higher priority than the power operator.

The order of precedence of the operators beginning with the highest is:

\ ¬ − +	(prefix operators)
**	(power)
* / % //	(multiply and divide)
+ −	(add and subtract)
(blank) ¦¦ (abuttal)	(concatenation with or without blank)
= > <	(comparison operators)
= = > > < <	(comparison operators)
\= ¬ =	(comparison operators)
> < < >	(comparison operators)
\> ¬ >	(comparison operators)
\< ¬ <	(comparison operators)
\= = ¬ = =	(comparison operators)
\> > ¬ > >	(comparison operators)
\< < ¬ < <	(comparison operators)
> = > > =	(comparison operators)
< = < < =	(comparison operators)
&	(AND)
¦ &&	(OR, exclusive-OR)

Examples

Suppose that the following symbols represent variables;

A has the value 3 and DAY has the value Monday.

Then:

```
A+5                     ->      '8'
A-4*2                   ->      '-5'
A/2                     ->      '1.5'
0.5**2                  ->      '0.25'
(A+1)>7                 ->      '0'         /* that is, False */
' '='                   ->      '1'         /* that is, True  */
' '=='                  ->      '0'         /* that is, False */
' '¬=='                 ->      '1'         /* that is, True  */
(A+1)*3=12              ->      '1'         /* that is, True  */
Today is Day            ->      'TODAY IS Monday'
'If it is' day          ->      'If it is Monday'
Substr(Day,2,3)         ->      'ond'       /* Substr is a function */
'!'xxx'!'               ->      '!XXX!'
'abc' << 'abd'          ->      '1'         /* that is, True  */
'077' >> '11'           ->      '0'         /* that is, False */
'abc' >> 'ab'           ->      '1'         /* that is, True  */
'ab ' << 'abd'          ->      '1'         /* that is, True  */
```

Note: The Procedures Language order of precedence usually causes no difficulty because it is the same as in conventional algebra and other computer languages. There are two differences from common notations:

- The prefix minus operator always has a higher priority than the power operator.
- Power operators (like other operators) are evaluated left-to-right.

For example:

```
-3**2      == 9  /* not -9  */
-(2+1)**2  == 9  /* not -9  */
2**2**3    == 64 /* not 256 */
```

Clauses and Instructions

Clauses can be subdivided into six types.

Null

A *null clause* is a clause consisting only of blanks and comments. A *null clause* is completely ignored (except that if it includes a comment, it is traced, if appropriate).

Note: A *null clause* is not an instruction; for example, putting an extra semicolon after the THEN or ELSE keywords in an IF instruction is not equivalent to using a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

Labels

A *label* is a clause that consists of a single symbol followed by a colon. The colon acts as an implicit clause terminator, so no semicolon is required. Labels are used to identify the targets of CALL instructions, SIGNAL instructions, and internal function calls. They can be traced selectively to aid debugging.

Any number of successive clauses may be labels, thus permitting multiple labels before another type of clause. Duplicate labels are permitted, but since the search effectively starts at the top of the program, the control, following a CALL or SIGNAL instruction, is always passed to the first occurrence of the label. The duplicate labels occurring later can be traced but cannot be used as a target of a CALL, SIGNAL, or function invocation.

Instructions

An *instruction* consists of one or more clauses describing some course of action for the language processor to take. Instructions can be: assignments, keyword instructions, or commands.

Assignments

An *assignment* is a single clause of the form **symbol=expression**. An assignment gives a variable a new value.

Keyword Instructions

A *keyword instruction* is one or more clauses, the first of which starts with a keyword that identifies the instruction. Keyword instructions control the external interfaces, the flow of control, and so on. Some instructions can include other nested instructions. In this example, the DO construct (DO, the group of instructions that follow it, and its associated END keyword) is considered a single keyword instruction.

```
DO
   instruction
   instruction
   instruction
END
```

Commands

A *command* is a single clause consisting of an expression only. The expression is evaluated and passed as a command string to some external environment.

Assignments and Symbols

A *variable* is an object whose value can change during the running of a REXX program. The process of changing the value of a variable is called *assigning* a new value to it. The value of a variable is a single character string, of any length, which may contain *any* characters.

You can assign a new value to variables with the ARG, PARSE, or PULL instructions, the VALUE built-in function, or the variable pool interface, but the most common way of changing the value of a variable is the assignment instruction itself. Any clause of the form:

```
symbol=expresssion
```

is taken to be an assignment. The result of `expression` becomes the new value of the variable named by the symbol to the left of the equal sign. For example:

```
/* Next line gives "FRED" the value "Frederic" */
Fred='Frederic'
```

The symbol naming the variable cannot begin with a digit (0-9) or a period. (Without this restriction on the first character of a variable name, you could redefine a number; for example, 3=4; would give a variable called 3 the value 4.)

You can use symbols in an expression even if you have not assigned them values, because they have a defined value at all times. A variable you have not assigned a value is *uninitialized*, and its value is the characters of the symbol itself, translated to uppercase (for example, a lowercase a-z becomes an uppercase A-Z). However, if it is a compound symbol, described under "Compound Symbols" on page 19, its value is the derived name of the symbol. For example:

```
/* If "Freda" has not yet been assigned a value, */
/* then next line gives "FRED" the value "FREDA" */
Fred=Freda
```

Symbols can be subdivided into four classes: constant symbols, simple symbols, compound symbols, and stems. Constant symbols cannot be assigned a value. Simple symbols can be used for variables where the name corresponds to a single value. Compound symbols and stems are used for more complex collections of variables, such as arrays and lists.

Constant Symbols

A *constant symbol* starts with a digit (`0 through 9`) or a period.

You cannot change the value of a constant symbol. It is simply the string consisting of the characters of the symbol (that is, with any lowercase alphabetic characters translated to uppercase).

These are constant symbols:

```
77
827.53
.12345
12e5        /* Same as 12E5 */
3D
```

Simple Symbols

A *simple symbol* contains no periods and does not start with a digit (`0` through `9`).

If the symbol has been assigned a value, it names a variable and its value is the value of that variable. If the symbol has not been assigned a value, its value is the characters of the symbol translated to uppercase.

These are simple symbols:

```
FRED
Whatagoodidea?     /* Same as WHATAGOODIDEA? */
?12
```

Compound Symbols

A *compound symbol* contains at least one period and at least two other characters. It cannot start with a digit or a period and, if there is only one period, the period cannot be the last character.

The name begins with a *stem* (that part of the symbol up to and including the first period), which is followed by parts of the name (delimited by periods) that are constant symbols, simple symbols, or null. (Do not use constant symbols with embedded signs because, with the stem prefixed, the whole is not a valid symbol.)

These are compound symbols:

```
FRED.3
Array.I.J
AMESSY..One.2.
```

Before the symbol is used (that is, at the time of reference), the values of any simple symbols (I, J, and One in the example) are substituted into the symbol, thus generating a new derived name. This derived name is then used in the same way as a simple symbol. That is, its value is, by default, the derived name or (if it has been used as the target of an assignment) its value is the value of the variable named by the derived name.

The substitution into the symbol that takes place permits arbitrary indexing (subscripting) of collections of variables that have a common stem. Note that the values substituted can contain *any* characters (including periods). Substitution is done only once.

To summarize, the derived name of a compound variable that is referred to by the symbol

`s0.s1.s2. --- .sn`

is given by

`d0.v1.v2. --- .vn`

where `d0` is the uppercase form of the symbol `s0` and `v1` through `vn` are the values of the constant or simple symbols `s1` through `sn`. Any of the symbols `s1` through `sn`

can be null. The values v1 through vn can also be null and can contain *any* characters (in particular, lowercase characters are not translated to uppercase, blanks are not removed, and periods have no special significance).

You can use compound symbols to set up arrays and lists of variables in which the subscript is not necessarily numeric. A useful application is to set up an array in which the subscripts are taken from the value of one or more variables, so effecting a form of associative memory ("content addressable").

Some examples, in the form of a small extract from a REXX program, follow:

```
a=3     /* assigns '3' to the variable 'A'  */
b=4               /*   '4'      to 'B'      */
c='Fred'          /*   'Fred'   to 'C'      */
a.b='Fred'        /*   'Fred'   to 'A.4'    */
a.fred=5          /*   '5'      to 'A.FRED' */
a.c='Bill'        /*   'Bill'   to 'A.Fred' */
c.c=a.fred        /*   '5'      to 'C.Fred' */
x.a.b='Annie'     /*   'Annie'  to 'X.3.4'  */
say  a  b  c  a.a  a.b  a.c  c.a  a.fred x.a.4
/* displays the string:                     */
/*    '3 4 Fred A.3 Fred Bill C.3 5 Annie'  */
```

Implementation maximum: The length of a variable name, before and after substitution, cannot exceed 250 characters.

Stems

A *stem* contains only one period, which is the last character. A stem cannot start with a digit or a period.

These are stems:

```
FRED.
A.
```

By default, the value of a stem is the characters of its symbol (that is, any alphabetic character translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

Further, when a stem is used as the target of an assignment, *all possible* compound variables whose names begin with that stem receive the new value, whether they previously had a value or not. Following the assignment, a reference to any compound symbol with that stem returns the new value until another value is assigned to the stem or to the individual variable.

For example:

```
hole.  = "empty"
hole.9 = "full"

say  hole.1  hole.mouse  hole.9

/* says "empty empty full" */
```

Thus, you can give a whole collection of variables the same value. For example:

```
total. = 0
do forever
   say "Enter an amount and a name:"
   pull amount name
   if datatype(amount)='CHAR' then leave
   total.name = total.name + amount
   end
```

Note: You can always obtain the value that has been assigned to the whole collection of variables by using the stem. However, this is not the same as using a compound variable whose derived name is the same as the stem.

For example:

```
total. = 0
null = ""
total.null = total.null + 5
say total. total.null              /* says "0 5" */
```

You can manipulate collections of variables, referred to by their stem, with the DROP and PROCEDURE instructions. `DROP FRED.` drops all variables with that stem (see page 36), and `PROCEDURE EXPOSE FRED.` exposes *all possible* variables with that stem (see page 49).

Notes:

1. When the ARG, PARSE, or PULL instruction, the VALUE built-in function, or the variable pool interface changes a variable, the effect is identical to an assignment. A stem used in a parsing template therefore sets an entire collection of variables.

2. Since an expression can include the operator =, and an instruction can consist purely of an expression, a possible ambiguity is resolved by the following rule: any clause that starts with a symbol and whose second token is (or starts with) an = symbol is an *assignment*, rather than an expression (or an instruction). This is not a restriction, since you can ensure the clause is processed as a command in several ways, such as by putting a null string before the first name or by enclosing the first part of the expression in parentheses.

   Similarly, if you unintentionally use a REXX keyword as the variable name in an assignment, this should not cause confusion. For example, the clause:

   ```
   Address='10 Downing Street';
   ```

   is an assignment, not an ADDRESS instruction.

Commands to External Environments

The base system for the language processor is assumed to include at least one active environment for processing commands. One of these environments is selected by default on entry to a REXX program. You can change the environment by using the ADDRESS instruction. You can find out the name of the current environment by using the ADDRESS built-in function. The environment selected depends on the caller; for example, if a REXX program is called from the OS/2 program, then the default environment is CMD. If called from an editor that accepts subcommands from the language processor, the default environment may be that editor.

A REXX program can issue commands—called *subcommands*—to other OS/2 application programs. For example, a REXX program written for a text editor can inspect a file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been executed as expected, and display messages to the user when appropriate.

An application that uses REXX as a macro language must register its environment with the REXX language processor. For a discussion of this mechanism, see "Subcommand Interface" on page 157.

To process commands using the current environment, use a clause of the form:

```
expression;
```

The expression is evaluated, resulting in a character string (which may be the null string), which is then prepared as appropriate and submitted to the host environment.

The environment then processes the command (which may have side-effects). It eventually returns control to the language processor after setting a *return code*. The language processor places this return code in the REXX special variable RC. For example, where the default environment is the OS/2 program, the sequence:

```
fname = "CHESHIRE"
exten = "CAT"
"ERASE" fname"."exten
```

would result in the string `ERASE CHESHIRE.CAT` being passed to the OS/2 program. Of course, the simpler expression:

```
"ERASE CHESHIRE.CAT"
```

would have the same effect in this case.

On return, the return code placed in RC will have the value 0 if the file CHESHIRE.CAT was erased and a nonzero value if, for example, the file could not be found in the current directory.

Because of the return codes, errors and failures in commands can affect REXX processing if a condition trap for ERROR or FAILURE is set to ON (see Chapter 7, "Conditions and Condition Traps"). They can also cause the command to be traced if TRACE E or TRACE F is set. TRACE Normal is the same as TRACE F and is the default (see page 60).

Note: Remember that the expression is evaluated before it is passed to the environment. You should enclose in quotes any part of the expression that is not to be evaluated. Some examples follow:

```
delete "*".lst          /* not "multiplied by" */

var.003 = anyvalue
type "var.003"          /* not a compound symbol */

w = any
dir"/w"                 /* not "divided by ANY" */
```

Chapter 3. Keyword Instructions

A *keyword instruction* is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control, while others provide services to the programmer. Some keyword instructions, such as DO, can include nested instructions.

In the syntax diagrams on the following pages, symbols (words) in capitals denote keywords; other words (such as expression) denote a collection of symbols as defined previously. Note, however, that the keywords are not case-dependent: the symbols *if*, *If*, and *iF* all have the same effect. Note also that you can usually omit most of the clause delimiters (;) shown because they are implied by the end of a line.

As explained on page 17, a keyword instruction is recognized *only* if its keyword is the first token in a clause and if the second token does not start with an = character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are recognized in the same situation. Note that any clause that starts with a keyword defined by REXX cannot be a command. A syntax error results if the keywords are not in their correct positions in a DO, IF, or SELECT instruction. (The keyword THEN is also recognized in the body of an IF or WHEN clause.) In other contexts, keywords are not reserved and can be used as labels or as the names of variables (though this is generally not recommended).

Certain other keywords, known as subkeywords, are reserved within the clauses of individual instructions. For example, the symbols VALUE and WITH are subkeywords in the ADDRESS and PARSE instructions respectively. For details, refer to the description of the respective instruction.

Blanks adjacent to keywords have no effect other than to separate the keyword from the subsequent token. One or more blanks following VALUE are required to separate the expression from the subkeyword in the following example:

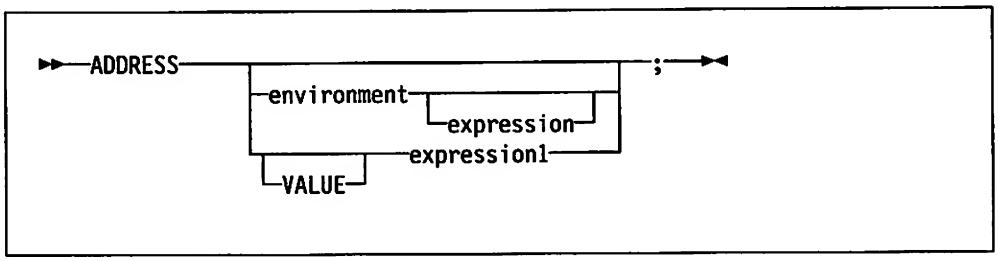
```
ADDRESS VALUE command
```

However, no blanks are required after the VALUE subkeyword in the following example, although they would add to the readability:

```
ADDRESS VALUE'ENVIR'||number
```

© Copyright IBM Corp. 1983, 1990

ADDRESS



ADDRESS temporarily or permanently changes the destination of commands.

The registration of alternative subcommand environments is described in "Subcommand Interface" on page 157.

To send a single command to a specified environment, code an *environment*, a literal string or a single symbol, which is taken to be a constant, followed by an *expression*. The *expression* is evaluated and the resulting command string is routed to *environment*. After the command is executed, *environment* is set back to whatever it was before, thus temporarily changing the destination for a single command.

Example:

```
ADDRESS CMD "DIR C:\STARTUP.CMD"   /*   OS/2   */

/* In a mainframe (e.g., CMS) environment, the  */
/* ADDRESS instruction might be used like this: */

ADDRESS CMS 'STATE PROFILE EXEC A'
```

If you specify only *environment*, a lasting change of destination occurs: all commands that follow (clauses that are neither REXX instructions nor assignment instructions) are routed to the specified command environment until the next ADDRESS instruction is executed. The previously selected environment is saved.

Example:

Suppose that the environment for a text editor is registered by the name EDIT:

```
address CMD
'DIR C:\STARTUP.CMD'
if rc=0 then 'COPY STARTUP.CMD *.TMP'
address EDIT
```

Subsequent commands are passed to the editor until the next ADDRESS instruction. The same would be true in a CMS system:

```
address CMS
'STATE PROFILE EXEC A'
if rc=0 then 'COPY PROFILE EXEC A TEMP = ='
address XEDIT
```

Similarly, you can use the VALUE form to make a lasting change to the environment. Here *expression1* (which may be just a variable name) is evaluated, and the result forms the name of the environment. You can omit the subkeyword

VALUE as long as *expression1* starts with a special character (so that it cannot be mistaken for a symbol or string).

Example:

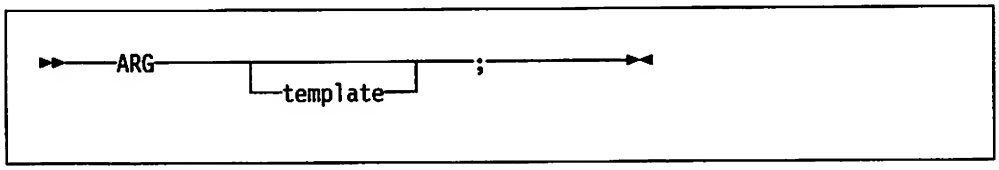
```
ADDRESS ('ENVIR'||number)
```

With no arguments, commands are routed back to the environment that was selected before the previous lasting change of environment was made, and the current environment name is saved. Repeated execution of ADDRESS alone therefore switches the command destination between two environments alternately. A null string for the environment name ("") is the same as the default environment.

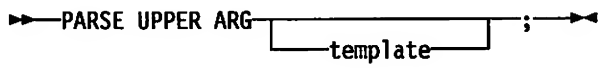
The two environment names are automatically saved across subroutine and internal function calls. See the CALL instruction on page 28 for more details.

You can retrieve the current ADDRESS setting using the ADDRESS built-in function described on page 72.

ARG



ARG retrieves the argument strings provided to a program or internal routine and assigns them to variables. It is a short form of the instruction:



The *template* is a list of symbols separated by blanks or patterns.

Unless a subroutine or internal function is being executed, the strings passed as parameters to the program are parsed into variables according to the rules described in the section on parsing.

If a subroutine or internal function is being executed, the data used will be the argument strings passed to the routine by the caller.

In either case, the strings being passed are translated to uppercase (that is, lowercase a-z to uppercase A-Z) before they are processed. Use the PARSE ARG instruction if you do not desire the uppercase translation.

The ARG (and PARSE ARG) instructions can be executed as often as desired (typically with different templates) and always parse the same current input strings. The only restrictions on the length or content of the data parsed are those the caller imposes.

Example:

```
/* String passed is "Easy Rider"  */

Arg adjective noun .

/* Now:  "ADJECTIVE"  contains 'EASY'          */
/*       "NOUN"       contains 'RIDER'         */
```

If you expect more than one string to be available to the program or routine, you can use a comma in the parsing template so each string is selected in turn.

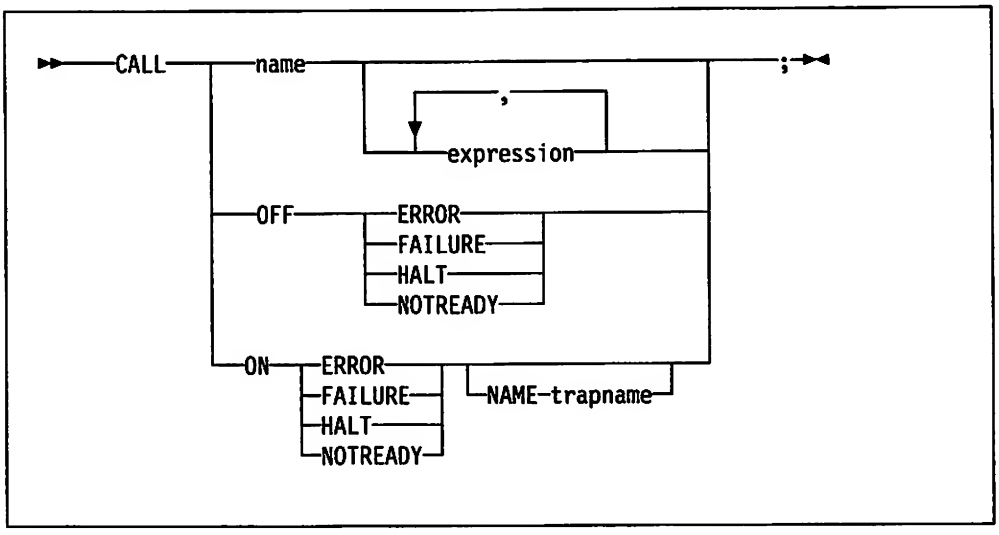
Example:

```
/* function is invoked by  FRED('data X',1,5)   */

Fred:  Arg string, num1, num2

/* Now:  "STRING" contains 'DATA X'            */
/*       "NUM1"   contains '1'                 */
/*       "NUM2"   contains '5'                 */
```

Notes:

1. The ARG built-in function can also retrieve or check the argument strings to a REXX program or internal routine. See page 72.
2. The source of the data being processed is also made available on entry to the program. See the PARSE instruction (SOURCE option) on page 48 for details.

CALL



CALL invokes a routine (if you specify *name*) or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap.

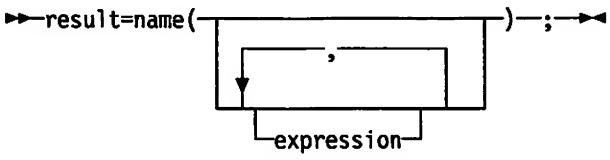
Note: For information on condition traps, see Chapter 7, "Conditions and Condition Traps."

To invoke a routine, specify *name*, a symbol or literal string that is taken as a constant. The *name* must be a symbol, which is treated literally, or a literal string. The routine invoked can be:

- An internal routine
- An external routine
- A built-in function.

If *name* is a string (that is, you specify *name* in quotes), the search for internal labels is bypassed and only a built-in function or an external routine is invoked. Note that the names of built-in functions (and generally the names of external routines) are in uppercase, and hence the name in the literal string should be in uppercase.

The invoked routine can optionally return a result so the CALL instruction is functionally identical to the clause:



except that the variable RESULT becomes uninitialized if the routine invoked returns no result.

The OS/2 program supports the specification of up to 20 expressions, separated by commas. The *expressions* are evaluated in order from left to right and form the argument strings during execution of the routine. Any ARG or PARSE ARG instructions or the ARG built-in function in the called routine accesses these strings, rather than those previously active in the calling program. You can omit expressions, if appropriate, by including extra commas.

The CALL then causes a branch to the routine called *name*, using exactly the same mechanism as function calls. Chapter 4, "Functions," describes the order in which these functions are searched for. A brief summary follows:

Internal routines
: These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If you specify the routine name in quotes, then an internal routine is not considered for that search order.

Built-in routines
: These are routines built in to the language processor for providing various functions. They always return a string containing the result of the function. (See page 70.)

External routines
: Users can write or use routines that are external to the language processor and the calling program. An external routine can be coded in any language, including REXX, that supports the system-dependent interfaces. If the CALL instruction invokes an external routine written in REXX as a subroutine, you can retrieve any argument strings with the ARG or PARSE ARG instructions or the ARG built-in function.

During execution of an internal routine, all variables previously known are normally accessible. However, the PROCEDURE instruction can set up a local variables environment to protect the subroutine and caller from each other. The EXPOSE option on the PROCEDURE instruction can expose selected variables to a routine.

Calling an external program as a subroutine is similar to calling an internal routine. The external routine, however, is an implicit PROCEDURE in that all the caller variables are always hidden and the status of internal values (NUMERIC settings, and so on) start with their defaults (rather than inheriting those of the caller).

When control reaches the internal routine, the line number of the CALL instruction is available in the variable SIGL (in the caller's variable environment). This can be used as a debug aid, as it is therefore possible to find out how control reached a routine. Note that if the internal routine uses the PROCEDURE instruction, then it needs to use the EXPOSE SIGL function to get access to the line number of the CALL.

Eventually a RETURN instruction should be executed by the subroutine and, at that point, control returns to the clause following the original CALL. If the RETURN instruction specified an expression, the variable RESULT is set to the value of that expression. Otherwise, the variable RESULT is dropped (becomes uninitialized).

An internal routine can include calls to other internal routines, as well as recursive calls to itself.

Example:

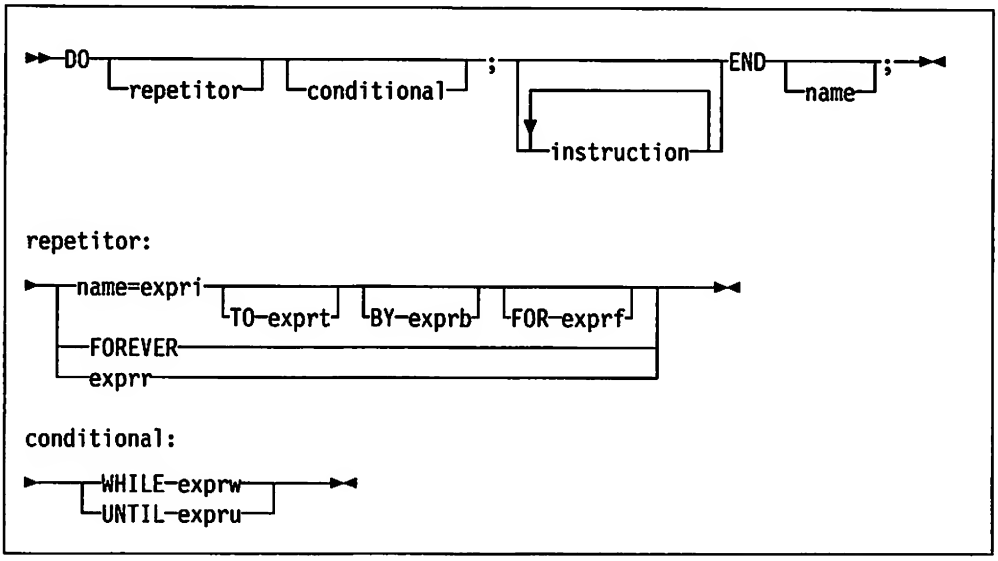
```
/* Recursive subroutine execution... */
arg x
call factorial x
say x'! =' result
exit

factorial: procedure     /* calculate factorial by..  */
  arg n                  /*  .. recursive invocation. */
  if n=0 then return 1
  call factorial n-1
  return  result * n
```

During internal subroutine (and function) execution, all important pieces of information are automatically saved and are then restored upon return from the routine. These pieces of information are:

- **The status of DO loops and other structures**—Executing a SIGNAL while within a subroutine is safe because DO loops that were active when the subroutine was called are not deactivated (but those currently active within the subroutine are deactivated).
- **Trace action**—Once a subroutine is debugged, you can insert a TRACE Off at the beginning of it; this does not affect the tracing of the caller. Conversely, if you only wish to debug a subroutine, you can insert a TRACE Results at the start and tracing is automatically restored to the conditions at entry (for example, Off) upon return. Similarly, ? (interactive debug) is saved across routines.
- **NUMERIC settings**—The DIGITS, FUZZ, and FORM of arithmetic operations, described on page 44, are saved and are then restored on return. A subroutine can therefore set the precision, and so on, that it needs to use without affecting the caller.
- **ADDRESS settings**—The current and secondary destinations for commands (see the ADDRESS instruction on page 24) are saved and are then restored on return.
- **Condition traps**—The CALL ON and SIGNAL ON are saved and then restored on return. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions the caller set up.
- **Condition information**—This is the information the CONDITION built-in function returns. See "CONDITION" on page 79.
- **Elapsed-time clocks**—A subroutine inherits the elapsed-time clock from its caller (see "TIME" on page 101), but since the time clock is saved across routine calls, a subroutine or internal function can independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.
- **OPTIONS ETMODE/EXMODE**—These option are saved and are then restored on return. For more information, see the OPTIONS instruction on page 46.

Implementation maximum: The total nesting of control structures, which includes internal routine calls, cannot exceed a depth of 100.

DO



DO groups instructions together and optionally executes them repetitively. During repetitive execution, a control variable (*name*) can be stepped through some range of values.

Syntax Notes:

- The *exprr, expri, exprb, exprt,* and *exprf* options (if present) are any expressions that evaluate to a number. The *exprr* and *exprf* options are further restricted to result in a non-negative whole number. If necessary, the numbers are rounded according to the setting of NUMERIC DIGITS.
- The *exprw* or *expru* options (if present) can be any expression that evaluates to 1 or 0.
- The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.
- The instructions can include assignments, commands, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO instruction itself).
- The subkeywords TO, BY, FOR, WHILE, and UNTIL are reserved within a DO instruction, in that they cannot be used as symbols in any of the expressions. FOREVER is also reserved, but only if it immediately follows the keyword DO.
- The *exprb* option defaults to 1, if relevant.

Simple DO Group

If you specify neither *repetitor* nor *conditional*, the construct groups a number of instructions together. These are executed once. Otherwise, the group of instructions is a *repetitive DO loop* and they are executed according to the repetitor phrase, optionally modified by the conditional phrase.

In the following example, the instructions are executed once.

Example:

```
/* The two instructions between DO and END will both */
/* be executed if A has the value 3.                 */
If a=3 then Do
            a=a+2
            Say 'Smile!'
            End
```

Simple Repetitive Loops

If *repetitor* is omitted but there is a *conditional option* or if the *repetitor* is FOREVER, the group of instructions is nominally executed forever, that is, until the condition is satisfied or a REXX instruction is executed that ends the loop (for example, LEAVE).

Note: For a discussion on conditional phrases, see "Conditional Phrases (WHILE and UNTIL)" on page 34.

In the simple form of a repetitive loop, *exprr* is evaluated immediately (and must result in a non-negative whole number) and the loop is then executed that many times.

Example:

```
/* This displays "Hello" five times */
Do 5
  say 'Hello'
  end
```

Note that, similar to the distinction between a command and an assignment, if the first token of *exprr* is a symbol and the second token is an = character, the controlled form of *repetitor* is expected.

Controlled Repetitive Loops

The controlled form specifies a *control variable*, *name*, which is assigned an initial value (the result of *expri*, formatted as though 0 had been added). The variable is then stepped (by adding the result of *exprb* at the bottom of the loop) each time the group of instructions is executed. The group is executed repeatedly if the end condition (determined by the result of *exprt*) is not met. If *exprb* is positive or 0, the loop is terminated when *name* is greater than *exprt*. If negative, the loop is terminated when *name* is less than *exprt*.

The *expri*, *exprt*, and *exprb* options must result in numbers. They are evaluated once only, before the loop begins and before the control variable is set to its initial value. The default value for *exprb* is 1. If *exprt* is omitted, the loop is executed indefinitely unless some other condition terminates it.

Example:

```
Do I=3 to -2 by -1          /* Would display:   */
  say i                     /*      3           */
  end                       /*      2           */
                            /*      1           */
                            /*      0           */
                            /*      -1          */
                            /*      -2          */
```

The numbers do not have to be whole numbers.

Example:

```
X=0.3                       /* Would display:   */
Do Y=X to X+4 by 0.7        /*      0.3         */
  say Y                     /*      1.0         */
  end                       /*      1.7         */
                            /*      2.4         */
                            /*      3.1         */
                            /*      3.8         */
```

The control variable can be altered within the loop; this may affect the iteration of the loop. Altering the value of the control variable is not normally considered good programming practice, though it may be necessary in certain circumstances.

Note that the end condition is tested at the start of each iteration (and, after the control variable is stepped, on the second and subsequent iterations). Therefore, if the end condition is met immediately, the group of instructions can be skipped entirely. Note also that the control variable is referred to by name. If, for example, the compound name "A.I" is used for the control variable, altering "I" within the loop causes a change in the control variable.

The execution of a controlled loop can be bounded further by a FOR phrase. In this case, you must specify *exprf*, and it must evaluate to a non-negative whole number. This acts in the same way as the repetition count in a simple repetitive loop and sets a limit to the number of iterations around the loop if no other condition terminates it. Similar to the TO and BY expressions, it is evaluated once only—when the DO instruction is first executed and before the control variable receives its initial value. Like the TO condition, the FOR condition is checked at the start of each iteration.

Example:

```
Do Y=0.3 to 4.3 by 0.7 for 3  /* Would display:    */
  say Y                       /*      0.3         */
  end                         /*      1.0         */
                              /*      1.7         */
```

In a controlled loop, the *name* describing the control variable can be specified on the END clause. This *name* must match *name* in the DO clause in all respects except case (note that no substitution for compound variables is carried out); a syntax error results if it does not. This enables the nesting of loops to be checked automatically, with minimal overhead.

Example:

```
Do K=1 to 10
  ...
  ...
  End k  /* Checks that this is the END for K loop */
```

Note: The NUMERIC settings can affect the successive values of the control variable since REXX arithmetic rules apply to the computation of stepping the control variable.

Conditional Phrases (WHILE and UNTIL)

A conditional phrase, which may cause termination of the loop, can follow any of the forms of *repetitor* (none, FOREVER, simple, or controlled). If you specify WHILE or UNTIL, *exprw* or *expru*, respectively, is evaluated each time around the loop using the latest values of all variables (and must evaluate to either 0 or 1). The group of instructions is repeatedly executed either while the result is 1 or until the result is 1.

For a WHILE loop, the condition is evaluated at the top of the group of instructions; for an UNTIL loop, the condition is evaluated at the bottom—before the control variable has been stepped.

Example:

```
Do I=1 to 10 by 2 until i>6
  say i
  end
/* Will display: 1, 3, 5, 7 */
```

Note: Using the LEAVE or ITERATE instructions can also modify the execution of repetitive loops.

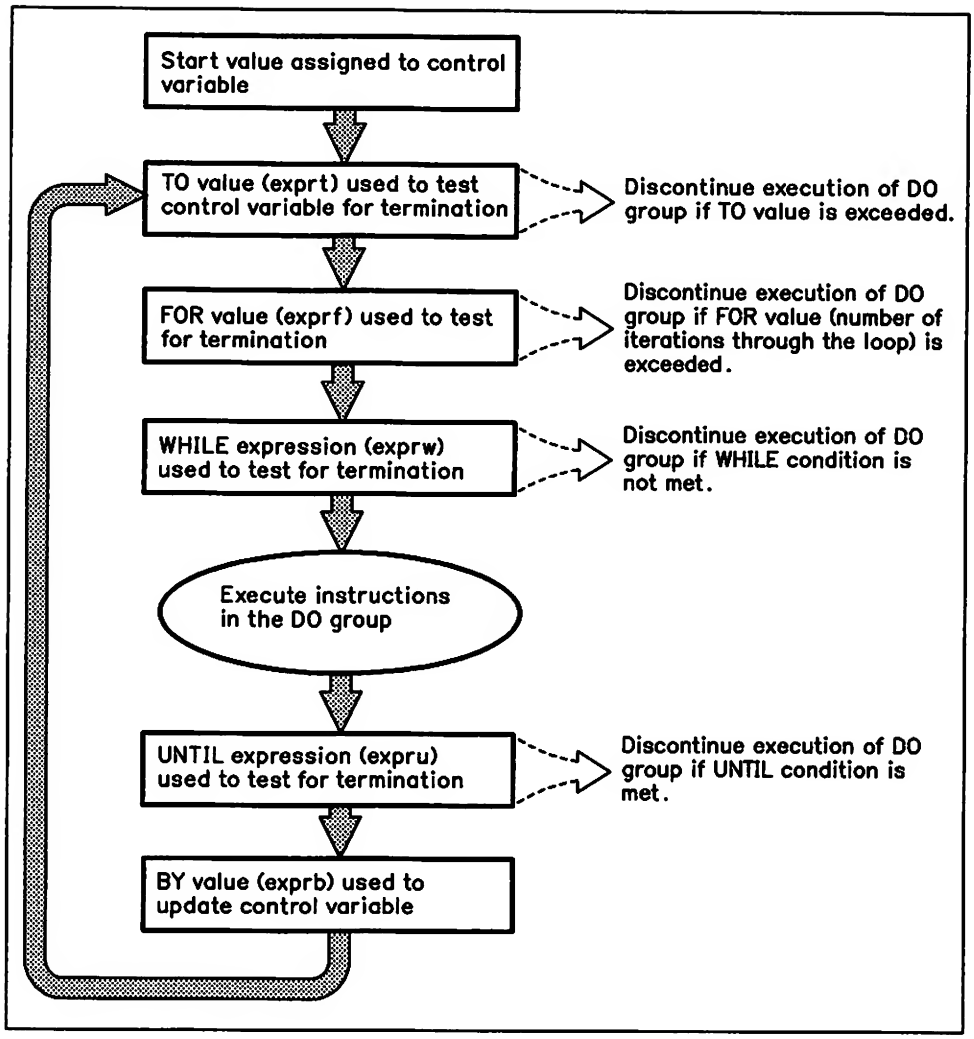
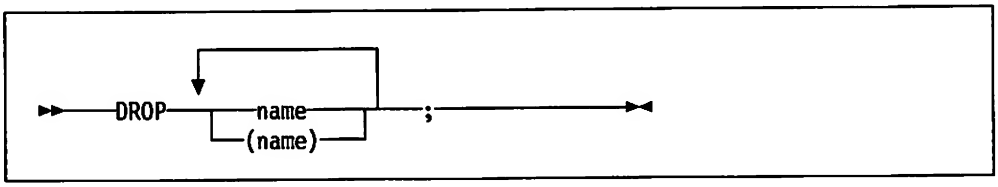


Figure 1. How a Typical DO Loop Is Executed

DROP



DROP *unassigns* variables, that is, restores them to their original uninitialized state. Each *name* identifies a variable you want to drop and must be a symbol that is a valid variable name, optionally enclosed in parentheses (to denote a subsidiary list) and separated from any other *name* by one or more blanks or comments.

Each variable specified is dropped from the list of known variables. If *name* is enclosed in parentheses (blanks are not necessary either inside or outside the parenthesis, but you can add them if desired), then its value is used as a subsidiary list of variables to drop. This stored list must follow the same rules as the main list (that is, valid variable names, separated by blanks, and so on) except that no parentheses are allowed. The variables are dropped in sequence from left to right. It is not an error to specify a name more than once or to DROP a variable that is not known. If an exposed variable is named (see "PROCEDURE" on page 49), the variable itself in the older generation is dropped.

Example:

```
j=4
Drop  a x.3 x.j
/* would reset the variables: "A", "X.3", and "X.4" */
/* so that reference to them returns their name.    */
```

Here, a variable name in parentheses is used as a subsidiary list.

Example:

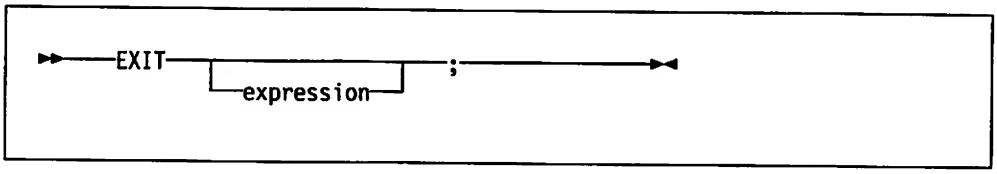
```
x=4;y=5;z=6;
a='x y z'
DROP (a)    /* will drop x,y, and z */
```

Specifying a stem (that is, a symbol that contains only one period) as the last character drops all variables starting with that stem.

Example:

```
Drop  x.
/* would reset all variables with names starting with "X." */
```

EXIT



EXIT leaves a program unconditionally. Optionally, EXIT returns a data string to the caller. The program is terminated immediately, even if an internal routine is currently being executed. If no internal routine is active, RETURN (see page 54) and EXIT are identical in their effect on the program that is being executed.

If you specify *expression*, it is evaluated and the string resulting from the evaluation is then passed back to the caller when the program terminates.

Example:

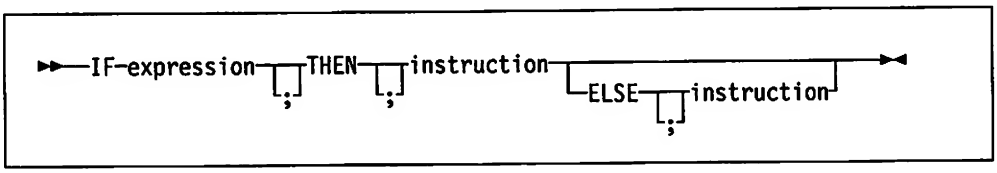
```
j=3
Exit j*4
/* Would exit with the string '12' */
```

If you do not specify *expression*, no data is passed back to the caller. If the program was called as an external function, this is detected as an error—either immediately (if RETURN was used), or on return to the caller (if EXIT was used).

"Running off the end" of the program is always equivalent to the instruction EXIT in that it terminates the whole program and returns no result string.

Note: The language processor does not distinguish between invocation as a command on the one hand and invocation as a subroutine or function on the other. If the program was invoked through a command interface, an attempt is made to convert the returned value to a return code acceptable by the host. (*Host* in this sense means the current command environment.) The returned string must be a whole number whose value will fit in a 16-bit signed integer (within the range −2**15 to 2**15−1). If the conversion fails, it is deemed to be a failure of the host interface (that is, the current command environment) and is thus not subject to trapping by SIGNAL ON SYNTAX.

IF



IF conditionally processes an instruction or group of instructions depending on the evaluation of the *expression*. The *expression* must evaluate to 0 or 1.

The instruction after the THEN clause is processed only if the result of the evaluation is 1. If you specify an ELSE clause, the instruction after ELSE is processed only if the result of the evaluation is 0.

Example:

```
if answer='YES' then say 'OK!'
                else say 'Why not?'
```

Remember that if the ELSE clause is on the same line as the last clause of the THEN part, you need a semicolon to terminate that clause.

Example:

```
if answer='YES' then say 'OK!';  else say 'Why not?'
```

The ELSE binds to the nearest IF at the same level. You can use the NOP instruction to eliminate errors and possible confusion when IF constructs are nested, as in the following example.

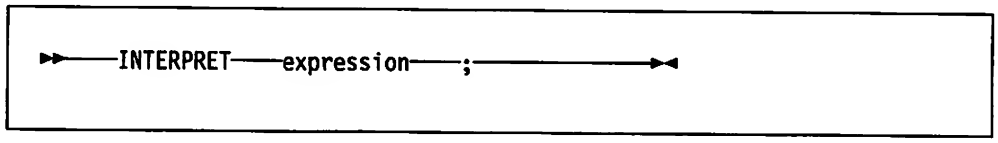
Example:

```
If answer = 'YES' Then
   If name = 'FRED' Then
      say 'OK, Fred.'
   Else
      nop
Else
   say 'Why not?'
```

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, SELECT, or the IF instruction itself. A null clause is not an instruction, so putting an extra semicolon after the THEN or ELSE clause is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.
2. The symbol THEN cannot be used within *expression* because the keyword THEN is treated differently in that it need not start a clause. This allows the expression on the IF clause to be terminated by the THEN clause, without a ; being required. If this were not so, users of other computer languages would experience considerable difficulties.

INTERPRET



INTERPRET executes instructions that have been built dynamically by evaluating *expression*.

The *expression* is evaluated and is then executed (interpreted) as though the resulting string was a line inserted into the input file (and bracketed by a DO; and an END; instruction).

Any instructions (including INTERPRET instructions) are allowed, but note that constructions such as DO ... END and SELECT ... END must be complete. For example, a string of instructions being interpreted cannot contain a LEAVE or ITERATE instruction (valid only within a repetitive DO loop) unless it also contains the whole repetitive DO ... END construct.

A semicolon is implied at the end of the expression during execution as a service to the user.

Example:

```
data='FRED'
interpret data '= 4'
/* Will a) build the string  "FRED = 4"          */
/*      b) execute  FRED = 4;                    */
/* Thus the variable "FRED" will be set to "4"  */
```

Example:

```
data='do 3; say "Hello there!"; end'
interpret data        /* Would display:         */
                      /*  Hello there!          */
                      /*  Hello there!          */
                      /*  Hello there!          */
```

Notes:

1. Labels within the interpreted string are not permanent and are therefore ignored. Hence, executing a SIGNAL instruction from within an interpreted string causes immediate exit from that string before the label search begins.
2. If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you may find that executing the instruction with TRACE *R* or TRACE *I* set is helpful.

Example:

If the program:

```
/* Here we have a small program. */
Trace Int
name='Kitty'
indirect='name'
interpret 'say "Hello"' indirect'"!"'
```

is run, the following trace is displayed.

```
[C:\]kitty
     3 *-* name='Kitty'
       >L>   "Kitty"
     4 *-* indirect='name'
       >L>   "name"
     5 *-* interpret 'say "Hello"' indirect'"!"'
       >L>   "say "Hello""
       >V>   "name"
       >O>   "say "Hello" name"
       >L>   ""!""
       >O>   "say "Hello" name"!""
       *-*  say "Hello" name"!"
       >L>    "Hello"
       >V>    "Kitty"
       >O>    "Hello Kitty"
       >L>    "!"
       >O>    "Hello Kitty!"
Hello Kitty!
[C:\]
```

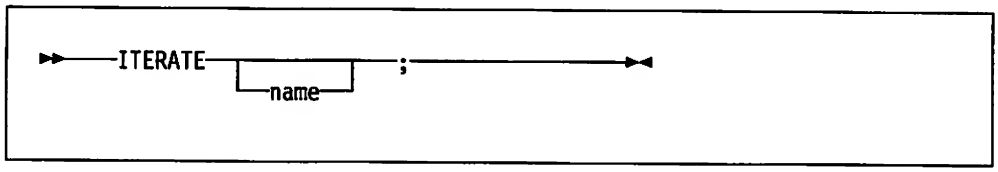
Lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up using a literal string, a variable (*INDIRECT*), and another literal. The resulting pure-character string is then interpreted as though it were actually part of the original program. Since it is a new clause, it is traced as such (the second *-* trace flag under line 5) and is then executed. Again, a literal string is concatenated to the value of a variable (*NAME*) and another literal, and the final result (`Hello Kitty!`) is then displayed.

3. For many purposes, the VALUE function (see page 104) can be used instead of the INTERPRET instruction. Line 5 in the preceding example could therefore have been replaced by:

   ```
   say "Hello" value(indirect)"!"
   ```

   INTERPRET is usually only required in special cases, such as when more than one statement is to be interpreted at once.

ITERATE



ITERATE alters the flow within a repetitive DO loop (that is, any DO construct other than that with a simple DO loop).

Execution of the group of instructions stops, and control is passed to the DO instruction as though the END clause had been encountered. The control variable (if any) is incremented and tested, as usual, and the group of instructions is executed again, unless the DO instruction terminates the loop.

If *name* is not specified, ITERATE steps the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop, which may be the innermost loop; this is the loop that is stepped. Any active loops inside the one selected for iteration are terminated (as though by a LEAVE instruction).

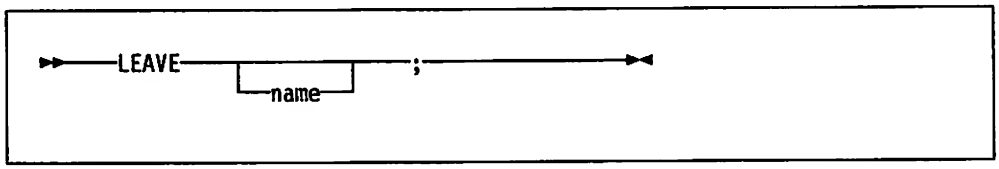
Example:

```
do i=1 to 4
  if i=2 then iterate
  say i
  end
/* Would display the numbers:   1, 3, 4  */
```

Notes:

1. If specified, *name* must match the name on the DO instruction in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being executed. If a subroutine is called (or an INTERPRET instruction is executed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to step an inactive loop.
3. If more than one active loop uses the same control variable, ITERATE selects the innermost loop.

LEAVE



LEAVE causes an immediate exit from one or more repetitive DO loops (that is, any DO construct other than a simple DO loop).

Processing of the group of instructions is terminated and control is passed to the instruction following the END clause as though the END clause had been encountered and the termination condition had been met normally. However, on exit, the control variable (if any) contains the value it had when the LEAVE instruction was processed.

If *name* is not specified, LEAVE terminates the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop, which may be the innermost loop; that loop (and any active loops inside it) is then terminated. Control then passes to the clause following the END clause that matches the DO clause of the selected loop.

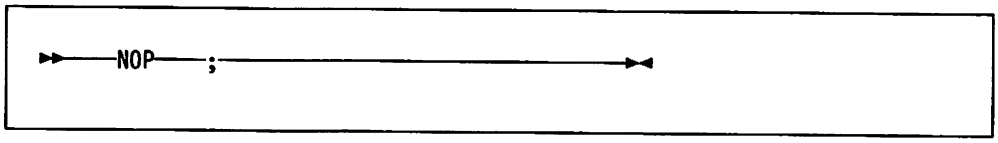
Example:

```
do i=1 to 5
  say i
  if i=3 then leave
  end
/* Would display the numbers:   1, 2, 3  */
```

Notes:

1. If specified, *name* must match the one on the DO instruction in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being executed. If a subroutine is called (or an INTERPRET instruction is executed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. LEAVE cannot be used to terminate an inactive loop.
3. If more than one active loop uses the same control variable, LEAVE selects the innermost one.

NOP



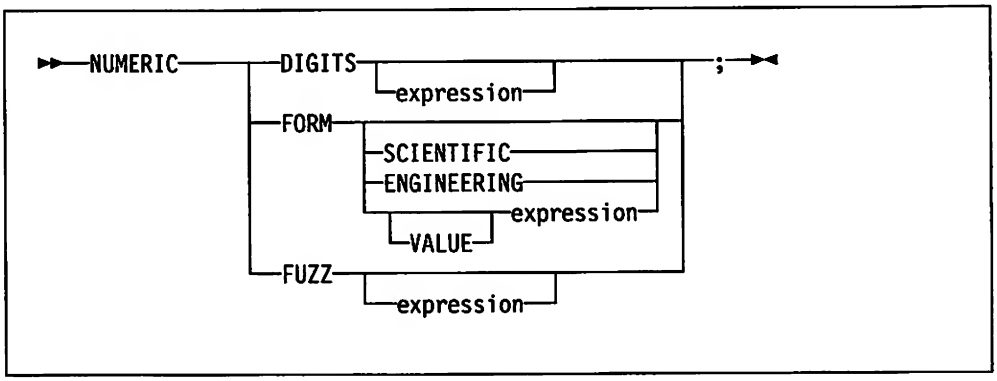
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause.

Example:

```
Select
   when a=b then nop           /* Do nothing */
   when a>b then say 'A > B'
   otherwise     say 'A < B'
end
```

Note: Putting an extra semicolon instead of the NOP inserts a null clause, which is ignored. The second WHEN clause is seen as the first instruction expected after the THEN clause, and hence is treated as a syntax error. NOP is a true instruction, however, and is a valid target for the THEN clause.

NUMERIC



NUMERIC changes the way in which arithmetic operations are carried out. The options of this instruction are described in detail on pages 125 through 134, but in summary:

NUMERIC DIGITS controls the precision to which arithmetic operations and arithmetic built-in functions are evaluated. If *expression* is omitted, the precision defaults to 9 digits. Otherwise, *expression* must evaluate to a positive whole number, rounded, if necessary, according to the current NUMERIC DIGITS setting and must be larger than the current NUMERIC FUZZ setting.

There is no limit to the value for DIGITS (except the amount of storage available), but note that high precisions are likely to require a good deal of processor time. It is recommended that you use the default value wherever possible.

You can retrieve the current NUMERIC DIGITS setting with the DIGITS built-in function. See "DIGITS" on page 84.

NUMERIC FORM controls the form of exponential notation REXX uses for the result of arithmetic operations and arithmetic built-in functions. This may be either SCIENTIFIC (in which case only one, nonzero digit appears before the decimal point) or ENGINEERING (in which case the power of 10 is always a multiple of three). The default is SCIENTIFIC. The FORM is set either directly by the subkeywords SCIENTIFIC or ENGINEERING or is taken from the result of evaluating the *expression* following VALUE. The result in this case must be either SCIENTIFIC or ENGINEERING. You can omit the subkeyword VALUE if the *expression* does not begin with a symbol or a literal string (for example, if it starts with a special character, such as an operator or parenthesis).

You can retrieve the current NUMERIC FORM setting with the FORM built-in function. See "FORM" on page 86.

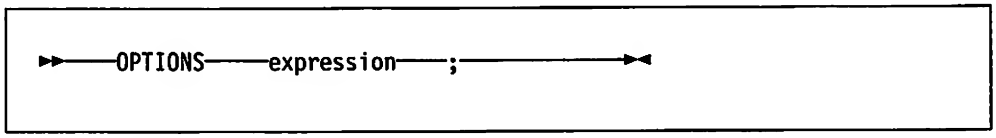
NUMERIC FUZZ controls the number of digits, at full precision, that are ignored during a numeric comparison operation. If *expression* is omitted, the default is 0 digits. Otherwise, *expression* must evaluate to 0 or a positive whole number, rounded, if necessary according to the current NUMERIC DIGITS setting and must be smaller than the current NUMERIC DIGITS setting.

FUZZ temporarily reduces the value of DIGITS by the FUZZ value before every numeric comparison operation. The numbers being compared are subtracted from each other under a precision of DIGITS minus FUZZ digits and this result is then compared with 0.

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function. See "FUZZ" on page 87.

Note: The three numeric settings are automatically saved across subroutine and internal function calls. See "CALL" on page 28 for more details.

OPTIONS



OPTIONS passes special requests or parameters to the language processor. For example, these may be language processor options or they may perhaps define a special character set.

The *expression* is evaluated, and the result is examined one word at a time. If the language processor recognizes the words, then they are obeyed. Words that are not recognized are ignored and assumed to be instructions to a different processor.

The language processors recognize the following words:

ETMODE Specifies that literal strings and comments containing DBCS characters are checked for being valid DBCS strings.

NOETMODE Specifies that literal strings and comments containing DBCS characters are not checked for being valid DBCS strings. NOETMODE is the default.

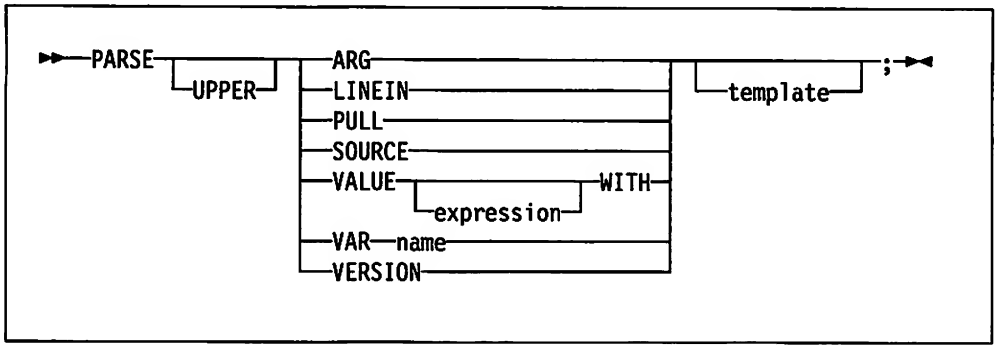
EXMODE Specifies that instructions, operators, and functions handle DBCS data in mixed strings on a logical character basis. DBCS data integrity is maintained.

NOEXMODE Specifies that any data in strings is handled on a byte basis. The integrity of DBCS characters, if any, may be lost. NOEXMODE is the default.

Notes:

1. Because of the scanning procedures of the language processor, you are advised to place an OPTIONS ETMODE instruction near the beginning of a program containing DBCS literal strings and comments.
2. To ensure proper scanning of a program containing DBCS literals and comments, type the words ETMODE, NOETMODE, EXMODE, and NOEXMODE as literal strings (that is, enclosed in quotes) in the OPTIONS instruction.
3. The OPTIONS ETMODE and OPTIONS EXMODE settings are saved and restored across subroutine and function calls.
4. The words ETMODE, EXMODE, NOEXMODE, and NOETMODE can occur several times within the result. The word that takes effect is determined by the last valid one specified between the pairs ETMODE-NOETMODE and EXMODE-NOEXMODE.

PARSE



PARSE assigns data (from various sources) to one or more variables according to the rules and templates described in Chapter 5, "Parsing for PARSE, ARG, and PULL."

If specified, a *template* is a list of symbols separated by blanks or patterns.

If you do not specify *template*, no variables are set but action is taken to get the data ready for parsing if necessary. Thus for PARSE PULL, a data string is removed from the current data queue; for PARSE LINEIN (and PARSE PULL if the current queue is empty), a line is taken from the default character input stream; and for PARSE VALUE, *expression* is evaluated. For PARSE VAR, the specified variable is accessed. If it does not have a value, the NOVALUE condition is raised, if it is enabled.

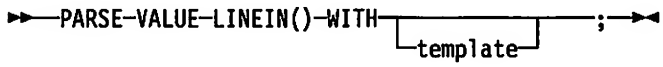
If you specify the UPPER option, the interpreter first translates the data to be parsed to uppercase (for example, a lowercase a-z becomes an uppercase A-Z). Otherwise, no uppercase translation takes place during the parsing.

The data used for each variant of the PARSE instruction is:

PARSE ARG—The strings passed to the program, subroutine, or function as the input argument list are parsed. (See "ARG" on page 26 for details and examples.)

Note: You can also retrieve or check the argument strings to a REXX program or internal routine with the ARG built-in function, described on page 72.

PARSE LINEIN—The next line from the default character input stream is parsed. (See Chapter 8, "Input and Output Streams," for a discussion of the REXX input model.) PARSE LINEIN is a shorter form of the instruction:



If no line is available, program execution normally pauses until a line is complete. Unlike PARSE PULL and PULL, the PARSE LINEIN instruction does not present the user with a question-mark prompt. Note that PARSE LINEIN should only be used when direct access to the character input stream is necessary. Normal line-by-line dialog with the user should be carried out with the PULL or PARSE PULL instructions to maintain generality and programmability.

To check if any lines are available in the default character input stream, use the built-in function LINES (see page 92). Also see page 89 for a description of the LINEIN function.

PARSE PULL—The next string from the queue is parsed. If the queue is empty, lines are read from the default input (typically the user's terminal). You can add data to the head or tail of the queue by using the PUSH and QUEUE instructions respectively. You can find the number of lines currently in the queue with the QUEUED built-in function, described on page 94. The queue remains active as long as the language processor is active. Other programs in the system can alter the queue and use it as a means of communication with programs written in REXX.

Note: PULL and PARSE PULL read first from the current data queue; if the queue is empty, they read from the default input stream, STDIN (typically, the keyboard). A question mark is displayed to the user as a prompt. (See the PULL instruction, on page 51, for further details.)

PARSE SOURCE—The data parsed describes the source of the program being executed.

The source string contains the characters OS/2, followed by either COMMAND, FUNCTION, or SUBROUTINE, depending on whether the program was invoked as a host command, invoked from a function call in an expression, or using the CALL instruction. These two tokens are followed by the complete path specification of the program file.

The string parsed might, therefore, be displayed as:

```
OS/2 COMMAND C:\OS2\REXTRY.CMD
```

PARSE VALUE— The *expression* is evaluated and the result is the data that is parsed. Note that WITH is a subkeyword in this context and cannot be used as a symbol within *expression*. For example:

```
PARSE VALUE time() WITH  hours ':' mins ':' secs
```

gets the current time and splits it up into its constituent parts.

PARSE VAR *name*—The value of the variable specified by *name* is parsed. The *name* must be a symbol that is valid as a variable name (that is, it cannot start with a period or a digit). Note that the variable *name* is not changed unless it is displayed in the template. For example:

```
PARSE VAR string word1 string
```

removes the first word from *string*, puts it in the variable *word1*, and assigns the remainder back to *string*. Similarly:

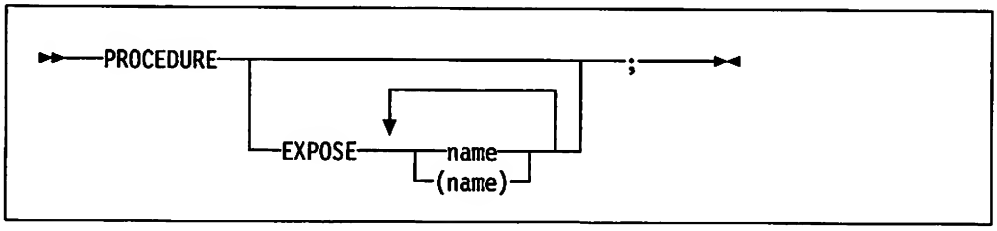
```
PARSE UPPER VAR string word1 string
```

also translates the data from *string* to uppercase before it is parsed.

PARSE VERSION—Information describing the language level and the date of the language processor is parsed. This consists of five words (delimited by blanks): first the string "REXXSAA", then the language-level description ("4.00"), and finally the release date ("13 June 1989").

Note: PARSE VERSION information should be parsed on a word basis rather than on an absolute column position basis.

PROCEDURE



PROCEDURE protects variables within an internal routine (subroutine or function) by making them unknown to the instructions that follow it. On executing a RETURN instruction, the original variables environment is restored and any variables used in the routine (which were not exposed) are dropped.

The EXPOSE option modifies this. Any variable specified by *name* is exposed, so that any reference to it (including setting and dropping) is made to the environment of the variables that the caller owns. With the EXPOSE option, you must specify at least one *name*, a symbol separated from any other *name* with one or more blanks. Optionally, you can enclose a *name* in parentheses to denote a subsidiary variable list. Any variables *not* specified by *name* on a PROCEDURE EXPOSE instruction are still protected. Hence, some limited set of the caller's variables can be made accessible and these variables can be changed (or new variables in this set can be created). All these changes are visible to the caller upon RETURN from the routine.

The variables are exposed in sequence from left to right. It is not an error to specify a name more than once or to specify a name that the caller has not used as a variable.

Example:

```
/* This is the main program */
j=1; x.1='a'
call toft
say j k m       /* would display "1 7 M" */
exit

toft: procedure expose j k x.j
   say j k x.j  /* would display "1 K a"   */
   k=7; m=3     /* note "M" is not exposed */
   return
```

Note that if *X.J* in the EXPOSE list had been placed before *J*, the caller value of *J* would not have been visible at that time, so *X.1* would not have been exposed.

If *name* is enclosed in parentheses (blanks are not necessary either inside or outside the parentheses but you can add them if desired) then, after that variable is exposed, the value of the variable is immediately used as a subsidiary list of variables that must follow the same rules as the main list (that is, valid variable names, separated by blanks, and so on) except that no parentheses are allowed. The variables name in a subsidiary list are also exposed from left to right.

Example:

```
/* This is the main program */
j=1;k=6;m=9
a ='j k m'
call test
exit
test:procedure expose (a)   /* Exposes A, J, K, and M  */
  say a j k m               /* Displays 'j k m 1 6 9'  */
  return
```

Specifying a *stem* in *names* exposes all possible compound variables whose names begin with that stem. (A *stem* is a symbol containing only one period, which is the last character. See page 20.)

Example:

```
lucky7:Procedure Expose i j a. b.
/* This exposes "I", "J", and all variables whose */
/* names start with "A." or "B."                  */
A.1='7'  /* This will set "A.1" in the caller's   */
         /* environment, even if it did not       */
         /* previously exist.                     */
```

Variables can be exposed through several generations of routines, if desired, by ensuring that they are included on all intermediate PROCEDURE instructions.

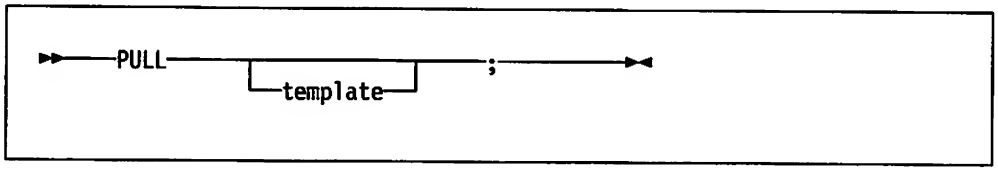
Only one PROCEDURE instruction in each level of routine call is allowed; all others (and those met outside of internal routines) are in error.

Notes:

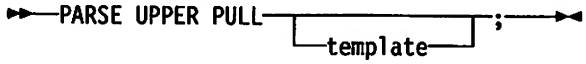
1. An internal routine need not include a PROCEDURE instruction, in which case the variables it is manipulating are those the caller owns.
2. The PROCEDURE instruction must be the first instruction executed after the CALL or function invocation—that is, it must be the first instruction following the label.

See the CALL instruction and function descriptions on pages 28 and 65 for details and examples of how routines are invoked.

PULL



PULL reads a string from the head of the currently active REXX data queue. It is a short form of the instruction:



The current head-of-queue is read as one string. Without a *template* specified, no further action is taken (and the string is thus effectively discarded). If specified, a *template* is a list of symbols separated by blanks or patterns. The string is translated to uppercase (for example, a lowercase a-z becomes an uppercase A-Z). and then parsed into variables according to the rules described in Chapter 5, "Parsing for PARSE, ARG, and PULL." Use the PARSE PULL instruction if you do not desire uppercase translation.

Note: If the current data queue is empty, PULL reads instead from STDIN (typically, the keyboard). A question mark is displayed to the user as a prompt. The length of data read by the PULL instruction is restricted to the length of strings contained by variables.

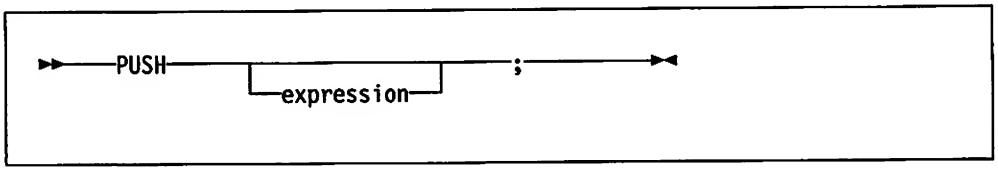
Example:

```
Say 'Do you want to erase the file?  Answer Yes or No:'
Pull answer .
if answer='NO' then Say 'The file will not be erased.'
```

Here the dummy placeholder "." is used on the template to isolate the first word the user enters.

The QUEUED built-in function, described on page 94, returns the number of lines currently in the queue.

PUSH



PUSH stacks the string resulting from the evaluation of *expression* in LIFO (last in, first out) format onto the currently active REXX data queue.

If you do not specify *expression*, a null string is stacked.

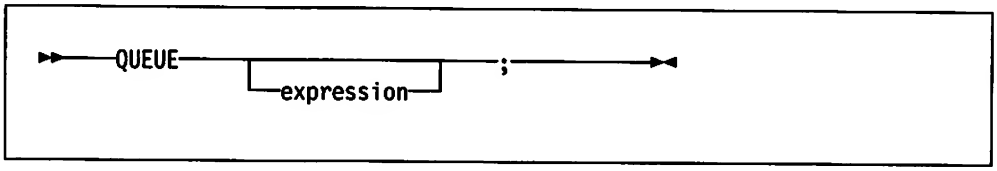
Note: Items placed on the current data queue have a maximum length of 64KB minus 64 bytes. Lines longer than this limit are truncated.

Example:

```
a='Fred'
push       /* Puts a null line onto the queue */
push a 2   /* Puts "Fred 2"    onto the queue */
```

The QUEUED built-in function, described on page 94, returns the number of lines currently in the queue.

QUEUE



QUEUE appends the string resulting from *expression* to the tail of the currently active REXX data queue. That is, it is added in FIFO (first in, first out) format.

If you do not specify *expression*, a null string is queued.

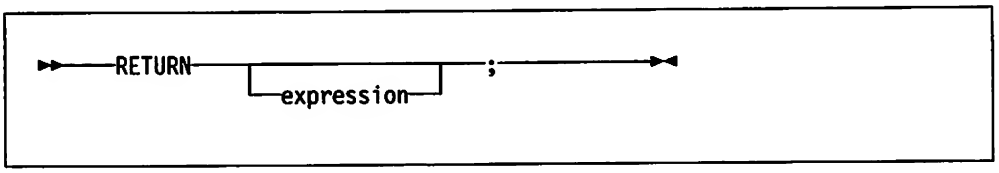
Note: Items placed on the current data queue have a maximum length of 64KB minus 64 bytes. Lines longer than this limit are truncated.

Example:

```
a='Toft'
queue a 2  /* Enqueues "Toft 2" */
queue      /* Enqueues a null line behind the last */
```

The QUEUED built-in function, described on page 94, returns the number of lines currently in the queue.

RETURN



RETURN returns control (and possibly a result) from a REXX program or internal routine to the point of its invocation.

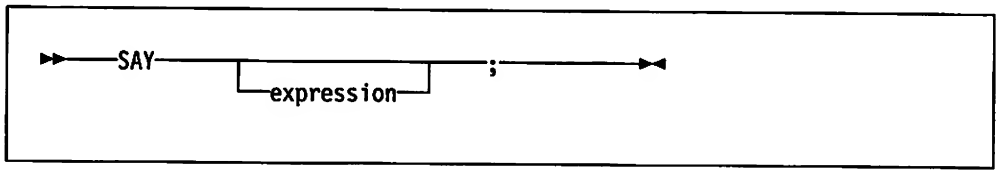
If no internal routine (subroutine or function) is active, RETURN and EXIT are identical in their effect on the program that is being executed. (See page 37.)

If a *subroutine* is being executed (see "CALL" on page 28), *expression* (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If *expression* is omitted, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, and so on) are also restored. (See page 28.)

If a *function* is being executed, the action taken is identical, except that *expression must* be specified on the RETURN instruction. The result of *expression* is then used in the original expression at the point where the function was invoked. See Chapter 4, "Functions," for more details.

If a PROCEDURE instruction was executed within the routine (subroutine or internal function), all variables of the current generation are dropped (and those of the previous generation are exposed) after *expression* is evaluated and before the result is used or assigned to RESULT.

SAY



SAY writes to the output stream the result of evaluating *expression*. The result is usually displayed to the user, but the output destination can depend on the implementation. The result of *expression* can be of any length.

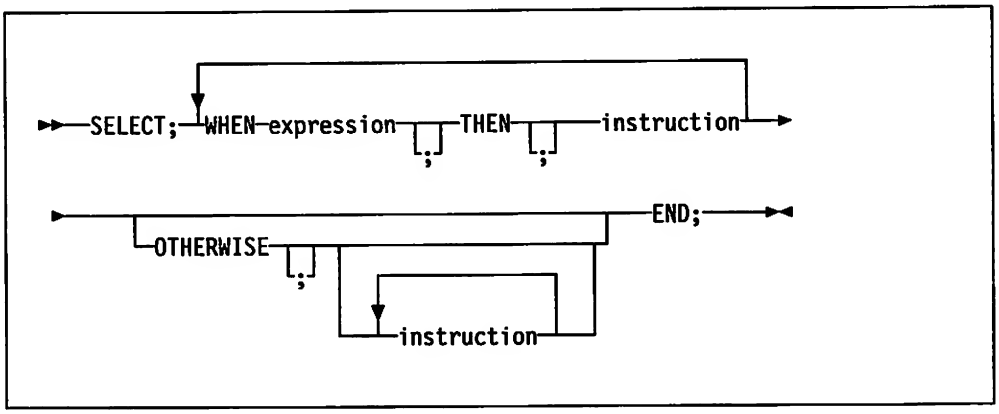
Notes:

1. Data from the SAY instruction is sent to the default output stream (STDOUT:) However, the standard OS/2 rules for redirecting output apply to SAY output; refer to the *User's Guide, Volume 1: Base Operating System.*
2. The SAY instruction does not format data; line wrapping is handled by the operating system and the hardware. Regardless of how formatting is accomplished, the output data remains a single logical line.

Example:

```
data=100
Say data 'divided by 4 =>' data/4
/* Would display: "100 divided by 4 => 25"  */
```

SELECT



SELECT conditionally executes one of several alternative instructions.

Each *expression* after a WHEN clause is evaluated in turn and must result in 0 or 1. If the result is 1, the instruction following the THEN clause (which may be a complex instruction such as IF, DO, or SELECT) is executed and control then passes to the END clause. If the result is 0, control passes to the next WHEN clause.

If none of the WHEN expressions evaluates to 1, control passes to the instructions, if any, after OTHERWISE. In this situation, the absence of OTHERWISE causes an error.

Example:

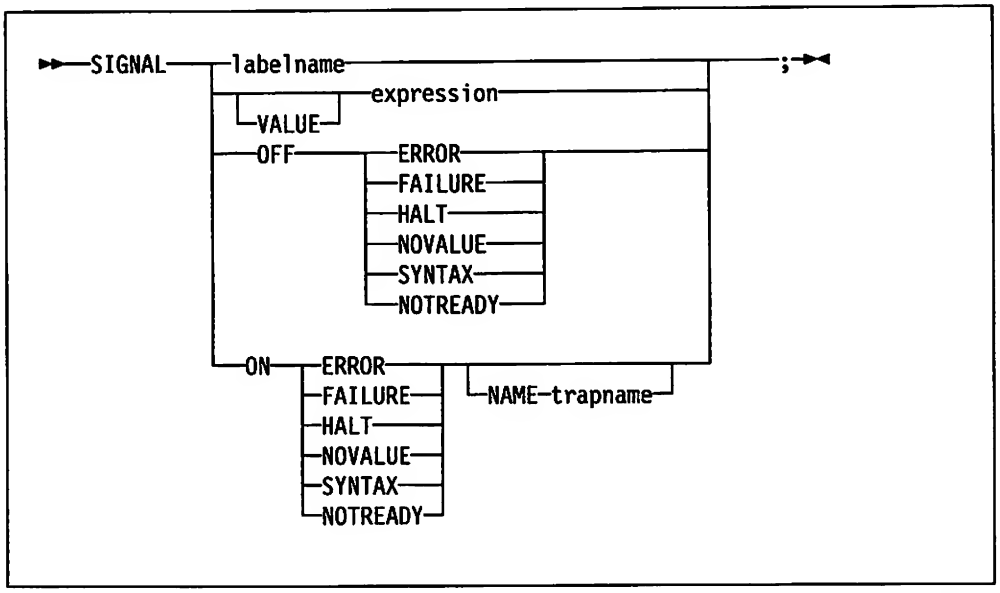
```
balance = balance - check
Select
  when balance > 0 then
       say 'Congratulations! You still have' balance 'dollars left.'
  when balance = 0 then do
       say 'Warning, Balance is now zero!  STOP all spending.'
       say "You cut it close this month! Hope you don't have any"
       say "checks left outstanding."
       end
  Otherwise
       say "You have just overdrawn your account."
       say "Your balance now shows" balance "dollars."
       say "Oops!  Hope the bank doesn't close your account."
end  /* Select */
```

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, IF, or the SELECT instruction itself.
2. A null clause is not an instruction, so putting an extra semicolon after a WHEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.
3. The symbol THEN cannot be used within *expression*. The keyword THEN is treated differently in that it need not start a clause. This allows the expression

on the WHEN clause to be terminated by the THEN without a ; (delimiter) being required.

SIGNAL



SIGNAL causes an *abnormal* change in the flow of control (if you specify *labelname*) or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap.

Note: For information on condition traps, see Chapter 7, "Conditions and Condition Traps."

To change the flow of control, a label name is derived from *labelname* or taken from the result of evaluating the *expression* after VALUE. The *labelname* you specify must be a symbol, treated literally, or a literal string that is taken as a constant. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or literal string (for example, if it starts with a special character, such as an operator or parenthesis). All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then terminated (that is, they cannot be reactivated). Control then passes to the first label in the program that matches the required string, as though the search had started from the top of the program.

Example:

```
Signal fred;  /* Jump to label "FRED" below */
  ....
  ....
Fred: say 'Hi!'
```

Because the search effectively starts at the top of the program, if duplicates are present, control always passes to the first occurrence of the label in the program.

When control reaches the specified label, the line number of the SIGNAL instruction is assigned to the special variable SIGL. This can aid debugging because SIGNAL can determine the source of a jump to a label.

Using SIGNAL with the INTERPRET Instruction

If, as the result of an INTERPRET instruction, a SIGNAL instruction is issued or a trapped event occurs, the remainder of the strings being interpreted are not searched for the given label. In effect, labels within interpreted strings are ignored.

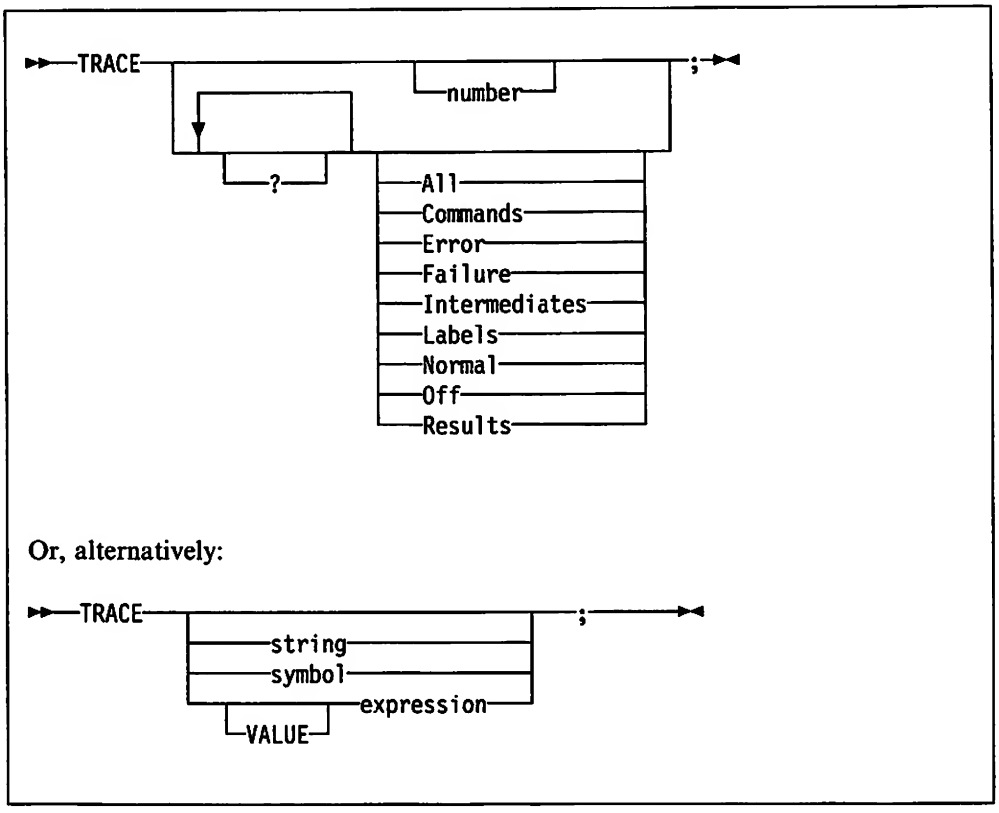
Using SIGNAL VALUE

The VALUE form of the SIGNAL instruction allows a branch to a label whose name is determined at the time of execution. This can safely effect a multiway CALL (or function call) to internal routines because any DO loops, and so forth, in the calling routine are protected against termination by the call mechanism.

Example:

```
fred='pete'
call multiway fred, 7
   ....
   ....
Multiway: procedure
   arg label .              /* One word, upper case                 */
                            /* Can add checks for valid labels here */
   signal value label       /* Jump to wherever                     */
   ....
Pete: say arg(1) '!' arg(2) /* Displays Pete ! 7                    */
   return
```

TRACE



Or, alternatively:

TRACE is primarily used for debugging. It controls the tracing action taken (that is, how much is displayed to the user) during execution of a REXX program. The syntax of TRACE is more concise than other REXX instructions. The economy of key strokes for this instruction is especially convenient since TRACE is usually entered manually during interactive debugging.

The *number* is a whole number.

The *string* or *expression* evaluates to:

- A number option
- The valid prefix, one of the alphabetic character (word) options described in the following text, or both
- Null.

The *symbol* is taken as a constant and is:

- A number option
- The valid prefix, one of the alphabetic character (word) options described in the following text, or both.

The option that follows TRACE or the result of evaluating *expression* determines the tracing action. If *expression* is used, you can omit the subkeyword VALUE as long as *expression* starts with a special character or operator (so it is not mistaken for a symbol or string).

Alphabetic Character (Word) Options

Although it is acceptable to enter the word in full, only the capitalized character is significant; all other letters are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

All — All clauses are traced (that is, displayed) before execution.

Commands — All host commands are traced before execution and any error return code is displayed.

Error — Any host command resulting in an error return code is traced after execution.

Failure — Any host command resulting in a failure is traced after execution. This is the same as the `Normal` option.

Intermediates — All clauses are traced before execution. Intermediate results during evaluation of expressions and substituted names are also traced.

Labels — Labels passed during execution are traced. This is especially useful with debug mode, when the language processor pauses after each label. It is also convenient for the user to make note of all subroutine calls and signals.

Normal — Any failing host command is traced after execution. *This is the default setting.*

For the default CMD processor in the OS/2 program, an attempt to issue an unknown command raises a FAILURE condition. An attempt to issue a command to an unknown subcommand environment also raises a FAILURE condition; in such a case, the variable RC is set to 2, the OS/2 return code for "file not found."

Off — Nothing is traced and the special prefix actions following are reset to OFF.

Results — All clauses are traced before execution. Final results (contrast with `Intermediates`, preceding) of evaluating an expression are traced. Values assigned during PULL, ARG, and PARSE instructions are also displayed. *This setting is recommended for general debugging.*

Prefix Option

The prefix ? is valid either alone or with one of the alphabetic character options. You can specify the prefix more than once, if desired. Each occurrence of a prefix on an instruction reverses the action of the previous prefix. The prefix must immediately precede the option (no intervening blanks).

The prefix ? modifies tracing and execution. ? is used to control interactive debug. During normal execution, a TRACE instruction prefixed with ? causes interactive debug to be switched on. (See Chapter 10, "Debugging Aids," for full details of this facility). While interactive debug is on, interpretation pauses after most clauses that are traced. For example, the instruction `TRACE ?E` makes the language processor pause for input after executing any host command that returns an error (that is, a nonzero return code).

Any TRACE instructions in the file being traced are ignored. (This is so that you are not taken out of interactive debug unexpectedly.)

When interactive debug is in effect, you can switch it off by issuing a TRACE instruction with a prefix ?. Repeated use of the ? prefix, therefore, switches you alternately in and out of interactive debug. Or, you can turn off interactive debug at any time by issuing `TRACE 0` or `TRACE` with no options.

Numeric Options

If interactive debug is active *and* if the option specified is a positive whole number (or an expression that evaluates to a positive whole number), that number indicates the number of debug pauses to be skipped over. (See Chapter 10, "Debugging Aids," for further information.) However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses. For example, `TRACE -100` means that the next 100 clauses that would normally be traced are not, in fact, displayed. After that, tracing resumes as before.

If interactive debug is not active, numeric options are ignored.

Tracing Tips

1. If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N and interactive debug (?) is set to off.
2. You can retrieve the trace actions currently in effect by using the TRACE built-in function described in "TRACE" on page 103.
3. Comments in the source REXX program are not included in the trace output.
4. Commands traced before execution always have the final value of the command (that is, the string passed to the environment) and the clause generating it produced in the traced output.
5. Trace actions are automatically saved across subroutine and function calls. See "CALL" on page 28 for more details.

A Typical Example

One of the most common traces you will use is:

```
TRACE ?R
/* Interactive debug is switched on if it was off, */
/*  and tracing Results of expressions begins.     */
```

Format of TRACE Output

Every clause traced is displayed with automatic formatting (indentation) according to its logical depth of nesting and so on, and the results (if requested) are indented an extra two spaces and are enclosed in double quotes so that leading and trailing blanks are apparent.

The first clause traced on any line is preceded by its line number. If the line number is greater than 99999, it is truncated on the left and a prefix of ? indicates the truncation. For example, the line number 100354 is shown as ?00354.

All lines displayed during tracing have a 3-character prefix to identify the type of data being traced. The prefixes and their definitions are:

`*-*` Identifies the source of a single clause, that is, the data actually in the program.

`+++` Identifies a trace message. This can be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program (see the following paragraph).

`>>>` Identifies the result of an expression (for TRACE R), the value assigned to a variable during parsing, or the value returned from a subroutine call.

`>.>` Identifies the value assigned to a placeholder during parsing (see page 120).

The following prefixes are only used if Intermediates (TRACE I) are being traced:

`>C>` The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.

`>F>` The data traced is the result of a function call.

`>L>` The data traced is a literal (string, uninitialized variable, or constant symbol).

`>O>` The data traced is the result of an operation on two terms.

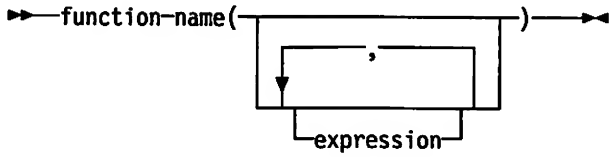
`>P>` The data traced is the result of a prefix operation.

`>V>` The data traced is the contents of a variable.

Following a syntax error that SIGNAL ON SYNTAX does not trap, the clause in error is always traced. If an attempt to transfer control to a label that could not be found caused the error, that label is also traced. The special trace prefix +++ identifies these traceback lines.

Chapter 4. Functions

You can include function calls to internal and external routines in an expression anywhere that a data term (such as a string) would be valid, using the notation:



function-name is a literal string or a single symbol that is taken to be a constant.

There can be up to an implementation maximum of expressions, separated by commas, between the parentheses. In the OS/2 program, the implementation maximum is 20 expressions. These expressions are called the *arguments* to the function. Each argument expression can include further function calls.

Note that the "(" must be adjacent to the name of the function, with no blank in between, or the construct is not recognized as a function call. (A *blank operator* is assumed at this point instead.)

The arguments are evaluated in turn from left to right and they are all then passed to the function. The function then executes some operation (usually dependent on the argument strings passed, though arguments are not mandatory) and eventually returns a single character string. This string is then included in the original expression as though the entire function reference had been replaced by the name of a variable that contained that data.

For example, the function SUBSTR is built into the language processor (see page 100) and could be used as:

```
N1='abcdefghijk'
Z1='Part of N1 is: 'Substr(N1,2,7)
/* would set Z1 to 'Part of N1 is: bcdefgh' */
```

A function call without any arguments must always include parentheses; otherwise, it would not be recognized as a function call.

```
date()  /* returns the date in the default format dd mon yyyy */
```

Calls to Functions and Subroutines

The function-calling mechanism is identical to the one for subroutines. The only difference between functions and subroutines is that functions must return data, whereas subroutines need not. The following types of routines can be called as functions:

Internal If the routine name exists as a label in the program, the current processing status is saved, so that it is later possible to return to the point of invocation to resume execution. Control is then passed to the first label in the program that matches the name. As with a routine invoked by the CALL instruction, various other pieces of status information (TRACE and NUMERIC settings and so on) are also saved. See the CALL instruction (page 28) for details about this. If you are calling an

© Copyright IBM Corp. 1983, 1990

internal routine as a function, you must specify an *expression* in any RETURN instruction to return from the function. This is not necessary if the function is called only as a subroutine.

Example:

```
/* Recursive internal function execution... */
arg x
say x'! =' factorial(x)
exit

factorial: procedure   /* calculate factorial by..  */
  arg n                /*  .. recursive invocation. */
  if n=0 then return 1
  return  factorial(n-1) * n
```

FACTORIAL is unusual in that it invokes itself (this is known as *recursive invocation*). The PROCEDURE instruction ensures that a new variable, n, is created for each invocation.

Built-in These functions are always available and are defined in the next section of this publication. (See pages 70 through 110.)

External You can write or make use of functions that are external to your program and to the language processor. An external function can be written in any language, including REXX, that supports the system-dependent interfaces the language processor uses to invoke it. Again, when called as a function it must return data to the caller.

Notes:

1. Calling an external REXX program as a function is similar to calling an internal routine. The external routine is, however, an implicit PROCEDURE in that all the caller variables are always hidden and the status of internal values (NUMERIC settings and so on) start with their defaults (rather than inheriting those of the caller).
2. Other REXX programs can be called as functions. You can use either EXIT or RETURN to leave the invoked REXX program; in either case, you must specify an expression.

Search Order

The search order for functions is the same as in the preceding list. That is, internal labels take first precedence, then built-in functions, and finally external functions.

Internal labels are *not* used if the function name is given as a string (that is, is specified in quotes); in this case, the function must be built-in or external. This lets you usurp the name of, for example, a built-in function to extend its capabilities, but still be able to invoke the built-in function when needed.

Example:

```
/* Modified DATE to return sorted date by default */
date: procedure
      arg in
      if in='' then in='Sorted'
      return 'DATE'(in)
```

Built-in functions have uppercase names. The name in the literal string must be in uppercase for the search to succeed, as in the example. The same is usually true of external functions.

External functions and *subroutines* have a system-defined search order.

REXX searches for external functions in the following order:

1. Functions that have been loaded into the macrospace for pre-order execution; see "Macrospace Interface" on page 176.
2. Functions that are part of a function package; see "External Functions" on page 167.
3. REXX functions in the current directory, with the current extension.
4. REXX functions along environment PATH, with the current extension.
5. REXX functions in the current directory, with the default extension.
6. REXX functions along environment PATH, with the default extension.
7. Functions that have been loaded into the macrospace for post-order execution.

The full search pattern for functions and routines is shown in Figure 2.

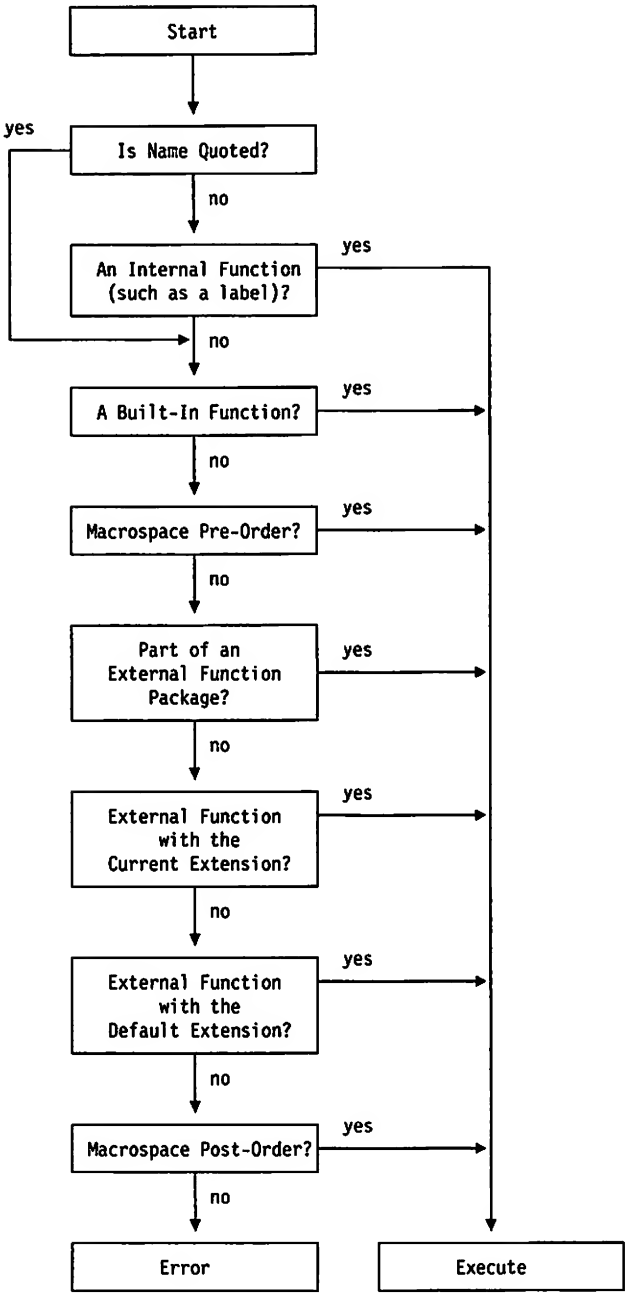


Figure 2. Function and Routine Resolution and Execution

Errors during Execution

If an external or built-in function detects an error of any kind, the language processor is informed and a syntax error results. Execution of the clause that included the function call is therefore terminated. Similarly, if an external function fails to return data correctly, the language processor detects this and reports it as an error.

If a syntax error occurs during the execution of an internal function, it can be trapped (using SIGNAL ON SYNTAX) and recovery may then be possible. If the error is not trapped, the program is terminated.

Return Values

A function normally returns a value that is substituted for the function call when the expression is evaluated.

The way the value returned by a function (or any REXX routine) is handled depends on whether it is called as a function call or called as a subroutine with the CALL instruction.

A routine called as a subroutine: If the routine returns a value, that value is stored in the special variable named RESULT. Otherwise, the RESULT variable is dropped, and its value is the string "RESULT".

A routine called as a function: If the function returns a value, that value is substituted into the expression at the position where the function was called. Otherwise, REXX stops with an error message.

Examples of different ways to call a REXX procedure follow:

```
call Beep 500, 100          /* Example 1: a subroutine call */
```

The built-in function BEEP is called as a REXX subroutine. The return value from BEEP is placed in the REXX special variable RESULT.

```
bc = Beep(500, 100)         /* Example 2: a function call   */
```

BEEP is called as a REXX function. The return value from the function is substituted for the function call. The clause itself is an assignment instruction; the return value from the BEEP function is placed in the variable bc.

```
Beep(500, 100)              /* Example 3: result passed as  */
                            /*            a command         */
```

The BEEP function is executed and its return value is substituted in the expression for the function call, just as in the preceding example. In this case, however, the clause as a whole evaluates to a single expression; therefore, the evaluated expression is passed to the current default environment as a command.

Note: Many other languages (such as C) throw away the return value of a function if it is not assigned to a variable. In REXX, however, a value returned as in the third example is passed on to the current environment or subcommand handler. If that environment is CMD (the default), then this action will result in the OS/2 program performing a disk search for what seems to be a command.

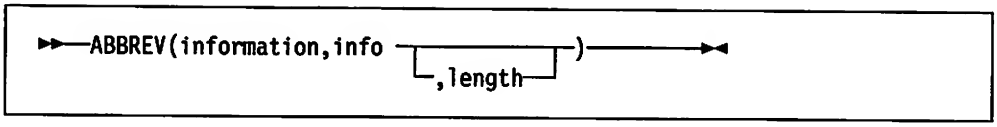
Built-In Functions

REXX provides a rich set of built-in functions. These include character manipulation, conversion, and information functions.

General notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the the name of the function with no space in between.
- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated.
- You can supply a null string where a *string* is referenced.
- If an argument specifies a length, it must be a non-negative whole number. If it specifies a start character or word in a string, it must be a positive whole number, unless otherwise stated.
- Where the last argument is optional, you can always include a comma to indicate you have omitted it; for example, DATATYPE(1,), like DATATYPE(1), would return NUM.
- If you specify a *pad* character, it must be exactly 1 character long.
- If a function has a suboption you can select by specifying the first character of a string, that character can be in uppercase or lowercase.
- Conversion between characters and hexadecimal involves the machine representation of character strings, and hence returns appropriately different results for ASCII and EBCDIC machines. The differences in output that result from EBCDIC-machine implementations are indicated, where appropriate, in the examples following.
- A number of the functions described in this chapter support the double-byte character set (DBCS). A complete list and description of these functions is given in Appendix B, "Double-Byte Character Set (DBCS)."

ABBREV (Abbreviate)



ABBREV returns 1 if *info* is equal to the leading characters of *information* and the length of *info* is not less than *length*. Returns 0 if neither of these conditions is met.

If you specify *length*, it must be a non-negative whole number. The default for *length* is the number of characters in *info*.

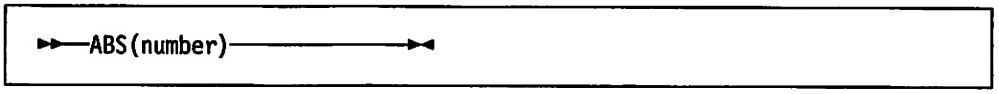
The following are some examples:

```
ABBREV('Print','Pri')      ->   1
ABBREV('PRINT','Pri')      ->   0
ABBREV('PRINT','PRI',4)    ->   0
ABBREV('PRINT','PRY')      ->   0
ABBREV('PRINT','')         ->   1
ABBREV('PRINT','',1)       ->   0
```

Note: A null string always matches if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired. For example:

```
say 'Enter option:';   pull option .
select  /* keyword1 is to be the default */
  when abbrev('keyword1',option) then ...
  when abbrev('keyword2',option) then ...
  ...
  otherwise nop;
end;
```

ABS (Absolute Value)

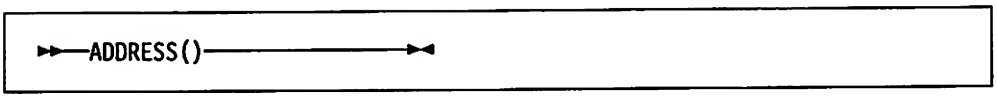


ABS returns the absolute value of *number*. The result has no sign and is formatted according to the current NUMERIC settings.

The following are some examples:

```
ABS('12.3')       ->   12.3
ABS(' -0.307')    ->   0.307
```

ADDRESS

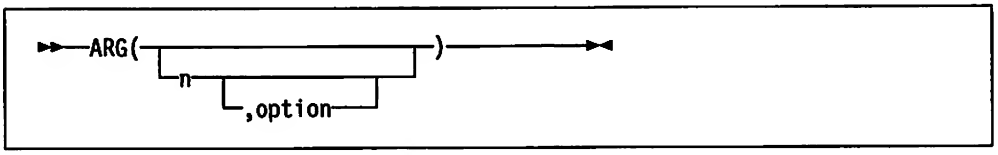


ADDRESS returns the name of the environment to which host commands are currently being submitted. Trailing blanks are removed from the result.

The following are some examples:

```
ADDRESS()    ->    'CMD'    /* OS/2 environment   */
ADDRESS()    ->    'EDIT'   /* possible editor    */
```

ARG (Argument)



ARG returns an argument string or information about the argument strings to a program or internal routine.

If you do not specify a parameter, the number of arguments passed to the program or internal routine is returned.

If you specify only *n*, the *n*th argument string is returned. If the argument string does not exist, the null string is returned. *n* must be a positive whole number.

If you specify *option*, ARG tests for the existence of the *n*th argument string. Valid options (of which only the capitalized letter is significant, all others are ignored) are:

Exists Returns 1 if the *n*th argument exists; that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.

Omitted Returns 1 if the *n*th argument was omitted; that is, if it was *not* explicitly specified when the routine was called. Returns 0 otherwise.

The following are some examples:

```
/*  following "Call name;" (no arguments) */
ARG()          ->    0
ARG(1)         ->    ''
ARG(2)         ->    ''
ARG(1,'e')     ->    0
ARG(1,'O')     ->    1

/*  following "Call name 'a',,'b';" */
ARG()          ->    3
ARG(1)         ->    'a'
ARG(2)         ->    ''
ARG(3)         ->    'b'
ARG(n)         ->    ''     /* for n>=4 */
ARG(1,'e')     ->    1
ARG(2,'E')     ->    0
ARG(2,'O')     ->    1
ARG(3,'o')     ->    0
ARG(4,'o')     ->    1
```

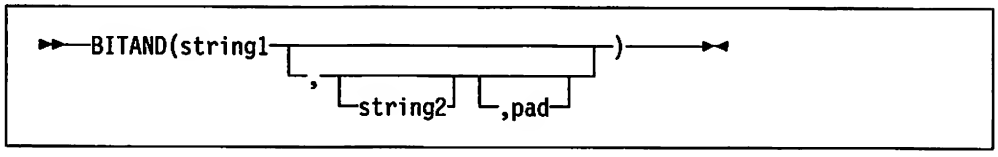
Notes:

1. You can retrieve and directly parse the argument strings to a program or internal routine with the ARG or PARSE ARG instructions. (See pages 26, 47, and 115.)
2. Programs called as commands either have one argument string or none. The program has no argument strings if it is called with the name only. However, the program has one argument string if anything else (including blanks) is included with the command.
3. Programs called by the REXSAA entry point (see page 154) can have multiple argument strings.

BEEP

This is an OS/2 program-specific function. See page 111.

BITAND (Bit by Bit AND)

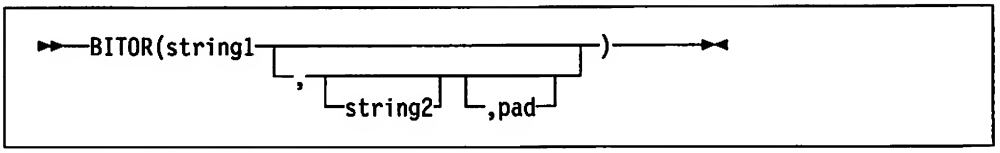


BITAND returns a string composed of the two input strings logically compared, bit by bit, using the AND operator. The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the AND operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it is used to extend the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero-length (null) string.

The following are some examples:

```
BITAND('73'x,'27'x)           ->   '23'x
BITAND('13'x,'5555'x)         ->   '1155'x
BITAND('13'x,'5555'x,'74'x)   ->   '1154'x
BITAND('pQrS',,'DF'x)         ->   'PQRS' /* ASCII only  */
BITAND('pQrS',,'BF'x)         ->   'pqrs' /* EBCDIC only */
```

BITOR (Bit by Bit OR)

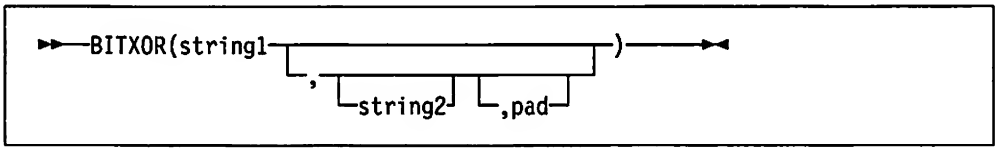


BITOR returns a string composed of the two input strings logically compared, bit by bit, using the OR operator. The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the OR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it is used to extend the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero-length (null) string.

The following are some examples:

```
BITOR('15'x,'24'x)            ->   '35'x
BITOR('15'x,'2456'x)          ->   '3556'x
BITOR('15'x,'2456'x,'F0'x)    ->   '35F6'x
BITOR('1111'x,,'4D'x)         ->   '5D5D'x
BITOR('pQrS',,'20'x)          ->   'pqrs' /* ASCII only  */
BITOR('pQrS',,'40'x)          ->   'PQRS' /* EBCDIC only */
```

BITXOR (Bit by Bit Exclusive OR)

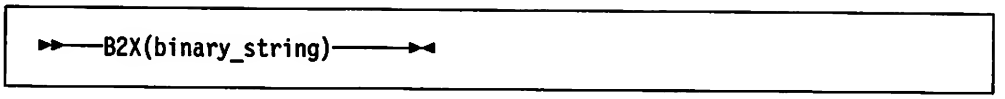


BITXOR returns a string composed of the two input strings logically compared, bit by bit, using the exclusive-OR operator. The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the XOR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it is used to extend the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero-length (null) string.

The following are some examples:

```
BITXOR('12'x,'22'x)             ->  '30'x
BITXOR('1211'x,'22'x)           ->  '3011'x
BITXOR('C711'x,'222222'x,' ')   ->  'E53362'x  /* EBCDIC */
BITXOR('C711'x,'222222'x,' ')   ->  'E53302'x  /* ASCII  */
BITXOR('1111'x,'444444'x)       ->  '555544'x
BITXOR('1111'x,'444444'x,'40'x) ->  '555504'x
BITXOR('1111'x,,'4D'x)          ->  '5C5C'x
```

B2X (Binary to Hexadecimal)



B2X returns a string, in character format, that represents *binary_string* converted to hexadecimal.

The *binary_string* is a string, of any length, of binary (0 or 1) digits. You can optionally include blanks in *binary_string* (at four-digit boundaries only, not leading or trailing) to aid readability; they are ignored.

The returned string uses uppercase alphabetics for the values A through F and does not include blanks.

If *binary_string* is null, B2X returns a null string. If the number of binary digits in *binary_string* is not a multiple of four, then up to three 0 digits are added on the left before the conversion to make a total that is a multiple of four.

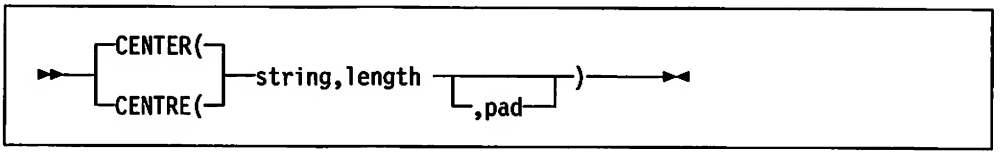
The following are some examples:

```
B2X('11000011')     ==   'C3'
B2X('10111')        ==   '17'
B2X('101')          ==   '5'
B2X('1 1111 0000') ==   '1F0'
```

You can combine B2X() with the functions X2D() and X2C() to convert a binary number into other forms. For example:

```
X2D(B2X('10111'))  ==   '23'   /* decimal 23 */
```

CENTER/CENTRE



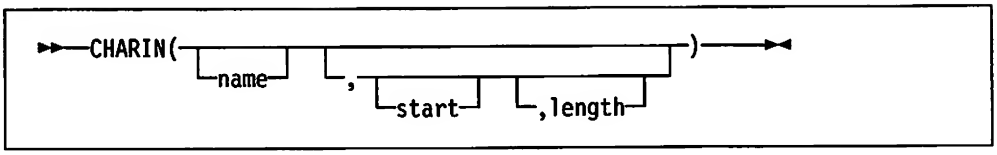
CENTER or CENTRE returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The default *pad* character is blank. If the string is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains 1 more character than the left-hand end.

The following are some examples:

```
CENTER(abc,7)              ->    '  ABC  '
CENTER(abc,8,'-')          ->    '--ABC---'
CENTRE('The blue sky',8)   ->    'e blue s'
CENTRE('The blue sky',7)   ->    'e blue '
```

Note: This function can be called either CENTER or CENTRE, which avoids errors due to the difference between the British and American spellings.

CHARIN (Characters of Input Read)



CHARIN returns a string of up to *length* characters read from the character input stream *name*. (See Chapter 8, "Input and Output Streams," for a discussion of the REXX input/output model.) The form of the *name* is implementation-dependent. If you omit *name*, characters are read from the device named STDIN:, which is the default input stream. The default *length* is 1.

For persistent streams, a read position is maintained for each stream. In the OS/2 program implementation, this is the same as the write position. Any read from the stream starts at the current read position by default. When the read is completed, the read position is increased by the number of characters read. You can give a *start* value to specify an explicit read position. This read position must be positive and within the bounds of the stream and must not be specified for a transient stream. A value of 1 for *start* refers to the first character in the stream.

If you specify a *length* of 0, then the read position is set to the value of *start* but no characters are read and the null string is returned.

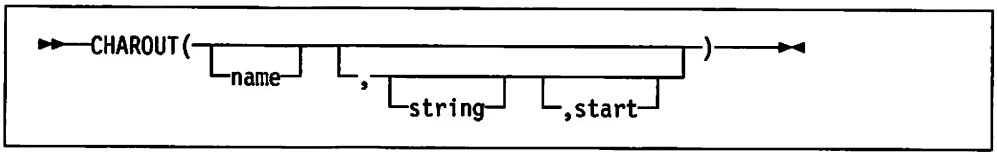
In a transient stream, if there are fewer then *length* characters available, then execution of the program normally stops until sufficient characters do become available. If, however, it is impossible for those characters to become available due to an error or other problem, the NOTREADY condition is raised (see "Errors During Input and Output" on page 147) and CHARIN returns with fewer than the requested number of characters.

The following are some examples:

```
CHARIN(myfile,1,3)   ->   'MFC'   /* the first 3     */
                                  /* characters      */
CHARIN(myfile,1,0)   ->   ''      /* now at start    */
CHARIN(myfile)       ->   'M'     /* after last call */
CHARIN(myfile,,2)    ->   'FC'    /* after last call */

/* Reading from the default input (here, the keyboard) */
/* User types 'abcd efg' */
CHARIN()             ->   'a'      /* default is  */
                                   /* 1 character */
CHARIN(,,5)          ->   'bcd e'
```

Note: When CHARIN is used to read from the keyboard, program execution stops until you press the Enter key. The line-feed and carriage-return characters of the Enter key are added to the end of the stream.

CHAROUT (Characters of Output to Write)



CHAROUT returns the count of characters remaining after attempting to write *string* to the character output stream *name*. (See Chapter 8, "Input and Output Streams," for a discussion of the REXX input/output model.) The form of the *name* is implementation-dependent. If you omit *name*, characters in *string* are written to the device STDOUT: (normally the display), which is the default output stream. *string* can be the null string, in which case no characters are written to the stream and 0 is always returned.

For persistent streams, a write position is maintained for each stream. In the OS/2 program implementation, this is the same as the read position. Any write to the stream starts at the current write position by default. When the write is completed, the write position is increased by the number of characters written. The initial write position is the end of the stream, so that calls to CHAROUT normally append to the end of the stream.

You can give a *start* value to specify an explicit write position for a persistent stream. This write position must be a positive whole number within the bounds of the stream (though it can specify the character position immediately after the end of the stream). A value of 1 for *start* refers to the first character in the stream.

Note: In some environments, overwriting a stream with CHAROUT or LINEOUT can erase (destroy) all existing data in the stream. This is not, however, the case in the OS/2 environment.

You can omit the *string* for persistent streams. In this case, the write position is set to the value of *start* that was given, no characters are written to the stream, and 0 is returned. If you do not specify *start* or *string*, the stream is closed. Again, 0 is returned.

Execution of the program normally stops until the output operation is effectively complete. If, however, it is impossible for all the characters to be written, the NOTREADY condition is raised (see "Errors During Input and Output" on page 147) and CHAROUT returns with the number of characters that could not be written (the residual count).

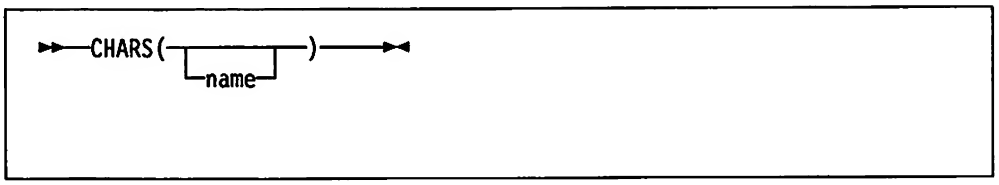
The following are some examples:

```
CHAROUT(myfile,'Hi')      ->   0   /* normally */
CHAROUT(myfile,'Hi',5)    ->   0   /* normally */
CHAROUT(myfile,,6)        ->   0   /* now at char 6 */
CHAROUT(myfile)           ->   0   /* at end of stream */
CHAROUT(,'Hi')            ->   0   /* normally */
CHAROUT(,'Hello')         ->   2   /* maybe */
```

Note: This routine is often called as a subroutine. The residual count is then available in the variable RESULT. For example:

```
Call CHAROUT myfile,'Hello'
Call CHAROUT myfile,'Hi',6
Call CHAROUT myfile
```

CHARS (Characters Remaining to Read)



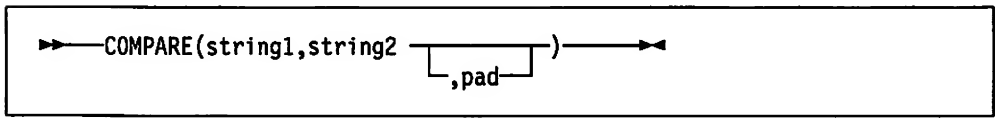
CHARS returns the total number of characters remaining in the character input stream *name*. The count includes any line separator characters, if these are defined for the stream, and, in the case of persistent streams, is the count of characters from the current read position. (See Chapter 8, "Input and Output Streams," for a discussion of the REXX input/output model.) The form of the *name* is implementation-dependent. If you omit *name*, the number of characters available in the default input stream (STDIN:) is returned.

The total number of characters remaining cannot be determined for some streams (for example, STDIN:). For these streams, the CHARS function returns 1 to indicate that data is present or 0 if no data is present. For OS/2 devices, CHARS always returns 1.

The following are some examples:

```
CHARS(myfile)     ->   42   /* perhaps */
CHARS(nonfile)    ->   0    /* perhaps */
CHARS()           ->   1    /* perhaps */
```

COMPARE

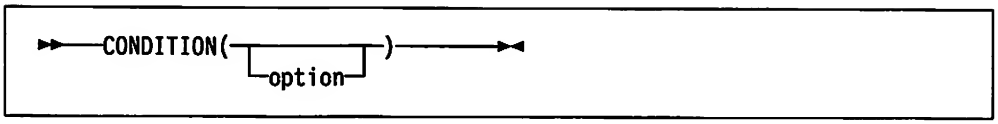


COMPARE returns 0 if the strings, *string1* and *string2*, are identical. Otherwise, COMPARE returns the position of the first character that does not match. The shorter string is padded on the right with *pad* if necessary. The default *pad* character is a blank.

The following are some examples:

```
COMPARE('abc','abc')          ->   0
COMPARE('abc','ak')           ->   2
COMPARE('ab ','ab')           ->   0
COMPARE('ab ','ab',' ')       ->   0
COMPARE('ab ','ab','x')       ->   3
COMPARE('ab-- ','ab','-')     ->   5
```

CONDITION



CONDITION returns the condition information associated with the current trapped condition. (See Chapter 7, "Conditions and Condition Traps," for a description of condition traps.) You can request four pieces of information:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition.

Request this information by using the following options (of which only the capitalized letter is needed, all others are ignored):

Condition name Returns the name of the current trapped condition.

Description Returns any descriptive string associated with the current trapped condition. See page 138 for the list of possible strings. If no description is available, returns a null string.

Instruction Returns either CALL or SIGNAL, the keyword for the instruction processed when the current condition was trapped. This is the default if you omit *option*.

Status Returns the status of the current trapped condition. This can change during processing, and is either:

ON The condition is enabled.

OFF The condition is disabled.

DELAY Any new occurrence of the condition is delayed.

If no condition has been trapped, then the CONDITION function returns a null string in all four cases.

The following are some examples:

```
CONDITION()          ->   'CALL'        /* perhaps */
CONDITION('C')       ->   'FAILURE'
CONDITION('I')       ->   'CALL'
CONDITION('D')       ->   'FailureTest'
CONDITION('S')       ->   'OFF'         /* perhaps */
```

Note: The CONDITION function returns condition information that is saved and restored across subroutine calls (including those a CALL ON condition trap causes). Therefore, once a subroutine invoked with CALL ON *trapname* has returned, the current trapped condition reverts to the condition before the CALL took place. CONDITION returns the values it returned before the condition was trapped.

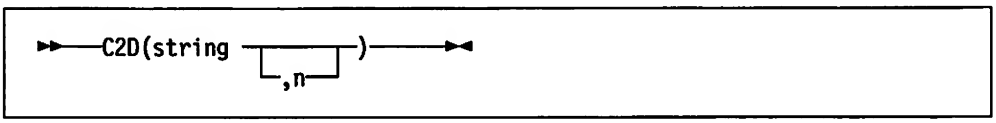
COPIES

COPIES returns *n* concatenated copies of *string*. *n* must be a non-negative whole number.

The following are some examples:

```
COPIES('abc',3)     ->    'abcabcabc'
COPIES('abc',0)     ->    ''
```

C2D (Character to Decimal)



C2D returns the decimal value of the binary representation of *string*. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS. If you specify *n*, it is the length of the returned result. If you do not specify *n*, *string* is processed as an unsigned binary number.

If *string* is null, `0` is returned.

The following are some examples:

```
C2D('09'X)       ->       9
C2D('81'X)       ->     129
C2D('FF81'X)     ->   65409
C2D('a')         ->     129     /*  EBCDIC  */
C2D('a')         ->      97     /* ASCII */
```

If you specify *n*, the given *string* is padded on the left with `'00'x` characters (note, not *sign-extended*) or truncated on the left to *n* characters. The resulting string of *n* hexadecimal digits is taken to be a signed binary number: positive if the leftmost bit is OFF, and negative, in two's complement notation, if the leftmost bit is ON. If *n* is `0`, C2D always returns `0`.

The following are some examples:

```
C2D('81'X,1)      ->    -127
C2D('81'X,2)      ->     129
C2D('FF81'X,2)    ->    -127
C2D('FF81'X,1)    ->    -127
C2D('FF7F'X,1)    ->     127
C2D('F081'X,2)    ->   -3967
C2D('F081'X,1)    ->    -127
C2D('0031'X,0)    ->       0
```

Implementation maximum: The input string cannot have more than 250 characters that are significant in forming the final result. Leading sign characters (`'00'x` and `'ff'x`) do not count towards this total.

C2X (Character to Hexadecimal)

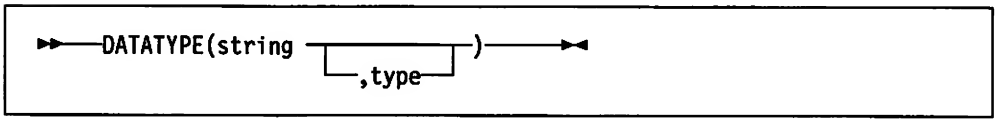
C2X returns a string, in character format, that represents *string* converted to hexadecimal. The returned string contains twice as many bytes as the input string because it is in literal string notation. For example, with an input of 1 (which is 'F1'X in EBCDIC), C2X returns F1, which is 'C6F1'X in EBCDIC. With an input of 1 (which is '31'X in ASCII), C2X returns 31, which is '3331'X in ASCII.

The string returned uses uppercase alphabetics for the values A through F and does not include blanks. If *string* is null, C2X returns a null string. The *string* can be any length.

The following are some examples:

```
C2X('0123'X)    ->    '0123'   /* '30313233'X     in ASCII */
C2X('ZD8')      ->    '5A4438' /* '354134343338'X in ASCII */
```

DATATYPE



DATATYPE returns NUM if you specify only *string* and if *string* is a valid REXX number (any format); returns CHAR if *string* is invalid.

If you specify *type*, DATATYPE returns 1 if *string* matches the type; otherwise DATATYPE returns 0. If *string* is null, DATATYPE returns 0 (except when *type* is X, which returns 1).

Request this information by using the following types (of which only the capitalized letter is needed, all others are ignored).

Alphanumeric Returns 1 if *string* contains only characters from the ranges a through z, A through Z, and 0 through 9.

Binary Returns 1 if *string* contains only the characters 0 or 1 or both.

C Returns 1 if *string* is a mixed SBCS/DBCS string.

Dbcs Returns 1 if *string* is a pure DBCS string.

Lowercase Returns 1 if *string* contains only characters from the range a through z.

Mixed case Returns 1 if *string* contains only characters from the ranges a through z and A through Z.

Number Returns 1 if *string* is a valid REXX number.

Symbol Returns 1 if *string* contains only characters that are valid in REXX symbols (see page 9). Note that both uppercase and lowercase alphabetics are permitted.

Uppercase Returns 1 if *string* contains only characters from the range A through Z.

Whole number Returns 1 if *string* is a REXX whole number under the current setting of NUMERIC DIGITS.

heXadecimal Returns 1 if *string* contains only characters from the ranges a through f, A through F, 0 through 9, and blank (as long as blanks appear only between pairs of hexadecimal characters). Hexadecimal also returns 1 if *string* is a null string.

The following are some examples:

```
DATATYPE(' 12 ')        ->   'NUM'
DATATYPE('')            ->   'CHAR'
DATATYPE('123*')        ->   'CHAR'
DATATYPE('12.3','N')    ->    1
DATATYPE('12.3','W')    ->    0
DATATYPE('Fred','M')    ->    1
DATATYPE('','M')        ->    0
DATATYPE('Fred','L')    ->    0
DATATYPE('?20K','S')    ->    1
DATATYPE('BCd3','X')    ->    1
DATATYPE('BC d3','X')   ->    1
```

DATE

DATE returns, by default, the local date in the format: dd mon yyyy (for example, 27 Aug 1988), with no leading zero or blank on the day. For *mon*, the first 3 characters of the English name of the month are used.

The following options (of which only the capitalized letter is needed, all others are ignored) obtain alternative formats:

Base Returns the number of complete days (not including the current day) since, and including, the base date, January 1, 0001, in the format: dddddd (no leading zeros). The expression `DATE(B)//7` returns a number in the range 0 through 6, where 0 is Monday and 6 is Sunday.

Note: The origin of January 1, 0001 is based on the Gregorian calendar. Though this calendar did not exist prior to 1582, the base date is calculated as if it did: 365 days per year, an extra day every four years except century years, and leap centuries if the century is divisible by 400. It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

Days Returns the number of days, including the current day, so far in this year in the format: ddd (no leading zeros).

European Returns date in the format: dd/mm/yy.

Language Returns date in an implementation- and language-dependent, or local, date format. In the OS/2 program, the language format is dd Month yyyy. If no local date format is available, the default format is returned.

Note: This format is intended to be used as a whole; REXX programs should not make any assumptions about the form or content of the returned string.

Month Returns full English name of the current month, for example, August.

Normal Returns date in the default format: dd mon yyyy.

Ordered Returns date in the format: yy/mm/dd (suitable for sorting, and so on).

Standard Returns date in the format: yyyymmdd (suitable for sorting, and so on).

Usa Returns date in the format: mm/dd/yy.

Weekday Returns the English name for the day of the week in mixed case. For example, Tuesday.

The following are some examples:

```
DATE()          ->    '27 Aug 1988'  /*  perhaps  */
DATE('B')       ->    725975
DATE('D')       ->    240
DATE('E')       ->    '27/08/88'
DATE('L')       ->    '27 August 1988'
DATE('M')       ->    'August'
DATE('N')       ->    '27 Aug 1988'
DATE('O')       ->    '88/08/27'
DATE('S')       ->    '19880827'
DATE('U')       ->    '08/27/88'
DATE('W')       ->    'Saturday'
```

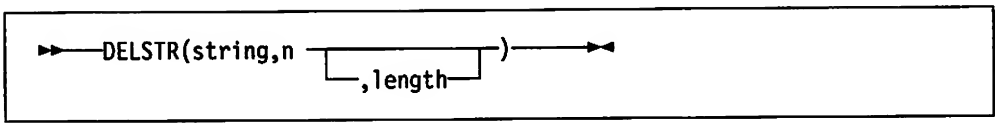
Note: The first call to DATE or TIME in one expression causes a time stamp to be made, which is then used for *all* calls to these functions in that expression. Hence, multiple calls to any of the DATE or TIME functions in a single expression are guaranteed to be consistent with each other.

DBCS

The following are all DBCS processing functions. See page 217.

DBADJUST	DBRIGHT	DBTOSBCS
DBBRACKET	DBRLEFT	DBUNBRACKET
DBCENTER	DBRRIGHT	DBVALIDATE
DBLEFT	DBTODBCS	DBWIDTH

DELSTR (Delete String)

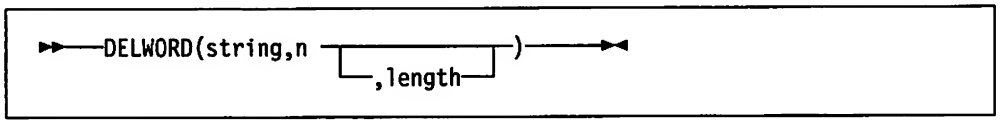


DELSTR returns *string* after deleting *length* characters beginning at the *n*th character. If you omit *length*, it defaults to the remaining characters in *string*. If *n* is greater than the length of *string*, returns string unchanged. *n* must be a positive whole number.

The following are some examples:

```
DELSTR('abcd',3)       ->    'ab'
DELSTR('abcde',3,2)    ->    'abe'
DELSTR('abcde',6)      ->    'abcde'
```

DELWORD (Delete Word)

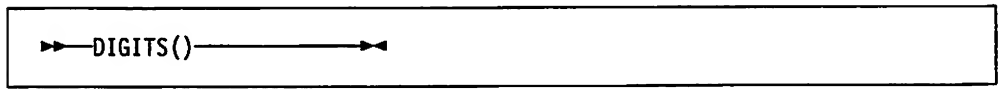


DELWORD returns *string* after deleting *length* blank-delimited words, beginning at the *n*th word. If you omit *length*, it defaults to the remaining words in *string*. *n* must be a positive whole number. If *n* is greater than the number of words in *string*, DELWORD returns *string* unchanged. The string deleted includes any blanks following the final word involved.

The following are some examples:

```
DELWORD('Now is the  time',2,2)  ->  'Now time'
DELWORD('Now is the time ',3)    ->  'Now is '
DELWORD('Now is the  time',5)    ->  'Now is the  time'
```

DIGITS



DIGITS returns the current setting of NUMERIC DIGITS.

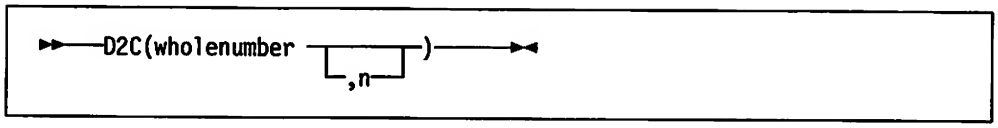
The following is an example:

```
DIGITS()    ->    9      /* by default */
```

DIRECTORY

This is an OS/2-specific function. See page 111.

D2C (Decimal to Character)



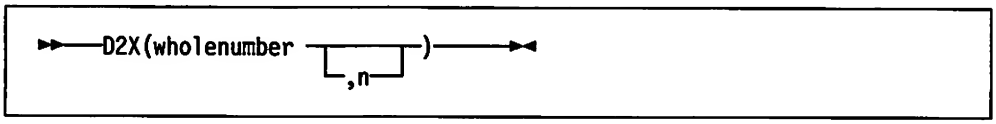
D2C returns a string, in character format, that is the ASCII representation of the decimal number. If you specify *n*, it is the length of the final result in characters. If you specify *n*, leading blanks are added to the output character.

If you omit *n*, *wholenumber* must be a non-negative number and the resulting length is as needed; therefore, the returned result has no leading '00'x characters.

The following are some examples:

```
D2C(65)      ->   'A'      /* '41'x is an ASCII 'A'    */
D2C(65,1)    ->   'A'
D2C(65,2)    ->   ' A'
D2C(65,5)    ->   '    A'
D2C(109)     ->   'm'      /* '6D'x  is an ASCII 'm'   */
D2C(-109,1)  ->   'ô'      /* '147'x is an ASCII 'ô'   */
D2C(76,2)    ->   ' L'     /* '4C'x  is an ASCII ' L'  */
D2C(-180,2)  ->   ' L'
```

D2X (Decimal to Hexadecimal)



D2X returns a string, in character format, that represents *wholenumber*, a decimal number, converted to hexadecimal. The returned string uses uppercase alphabetics for the values A through F and does not include blanks.

If you specify *n*, it is the length of the final result in characters. If you specify *n*, after conversion, the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, it is truncated on the left.

If you omit *n*, *wholenumber* must be a non-negative number and the returned result has no leading 0 characters.

The following are some examples:

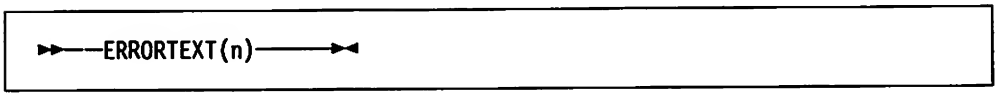
```
D2X(9)         ->   '9'
D2X(129)       ->   '81'
D2X(129,1)     ->   '1'
D2X(129,2)     ->   '81'
D2X(129,4)     ->   '0081'
D2X(257,2)     ->   '01'
D2X(-127,2)    ->   '81'
D2X(-127,4)    ->   'FF81'
D2X(12,0)      ->   ''
```

Implementation maximum: The output string cannot have more than 500 significant hexadecimal characters, though a longer result is possible if it has additional leading sign characters (0 and F).

ENDLOCAL

This is an OS/2-specific function. See page 112.

ERRORTEXT



ERRORTEXT returns the error message associated with error number *n*. The *n* must be in the range 0 through 99; any other value is an error. Returns the null string if *n* is in the allowed range but is not a defined REXX error number. See Appendix A, "Error Numbers and Messages," for a complete description of error numbers and messages.

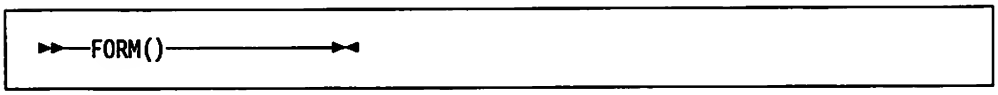
The following are some examples:

```
ERRORTEXT(16)    ->    'Label not found'
ERRORTEXT(60)    ->    ''
```

FILESPEC

This is an OS/2-specific function. See page 112.

FORM

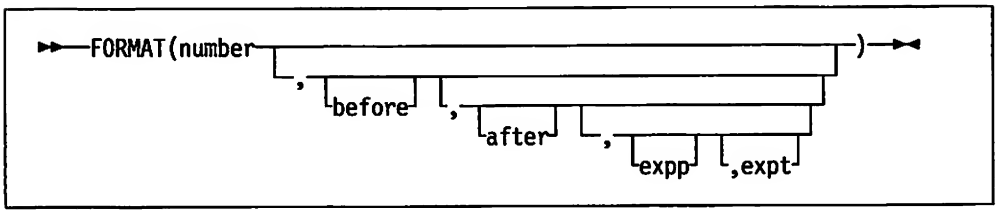


FORM returns the current setting of NUMERIC FORM.

The following is an example:

```
FORM()    ->    'SCIENTIFIC'  /* by default */
```

FORMAT



FORMAT returns *number*, rounded and formatted.

The *number* is first rounded and formatted to standard REXX rules, as though the operation *number+0* had been carried out. If you specify only *number*, the result is precisely that of this operation. If you specify any other options, the *number* is formatted as follows.

The *before* and *after* options describe how many characters are used for the integer part and decimal part of the result respectively. If you omit either or both of these, the number of characters used for that part is as needed.

If *before* is not large enough to contain the integer part of the number, an error results. If *before* is too large, the number is padded on the left with blanks. If *after* is not the same size as the decimal part of the number, the number is rounded (or extended with zeros) to fit. Specifying 0 causes the number to be rounded to an integer.

The following are some examples:

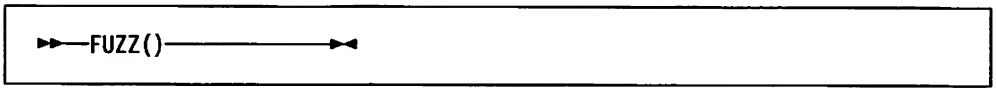
```
FORMAT('3',4)            ->    '   3'
FORMAT('1.73',4,0)       ->    '   2'
FORMAT('1.73',4,3)       ->    '   1.730'
FORMAT('-.76',4,1)       ->    '  -0.8'
FORMAT('3.03',4)         ->    '   3.03'
FORMAT(' - 12.73',,4)    ->    '-12.7300'
FORMAT(' - 12.73')       ->    '-12.73'
FORMAT('0.000')          ->    '0'
```

The first three arguments are as previously described. In addition, *expp* and *expt* control the exponent part of the result: *expp* sets the number of places for the exponent part; the default is to use as many as needed. The *expt* sets the trigger point for use of exponential notation. If the number of places needed for the integer part exceeds the value of *expt*, exponential notation is used. Likewise, exponential notation is used if the number of places needed for the decimal part exceeds twice the value of *expt*. The default is the current setting of NUMERIC DIGITS. If *expt* is 0, exponential notation is always used unless the exponent would be 0. If *expp* is 0, no exponent is supplied, and the number is expressed in *simple* form with added zeros as necessary (this overrides a 0 value of *expt*). Otherwise, if expp. is not large enough to contain the exponent, an error results. If the exponent would be 0 in this case (a non-zero *expp*), then *expp*+2 blanks are supplied for the exponent part of the result.

The following are some examples:

```
FORMAT('12345.73',,,2,2)     ->    '1.234573E+04'
FORMAT('12345.73',,3,,0)     ->    '1.235E+4'
FORMAT('1.234573',,3,,0)     ->    '1.235'
FORMAT('12345.73',,,3,6)     ->    '12345.73'
FORMAT('1234567e5',,3,0)     ->    '123456700000.000'
```

FUZZ

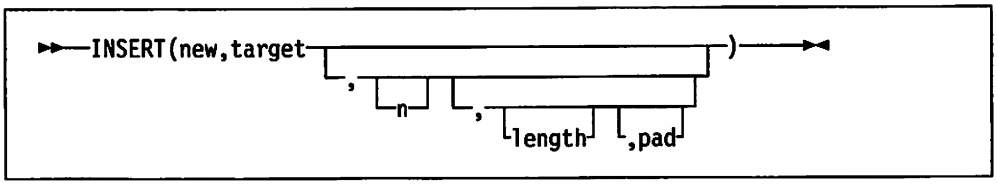


FUZZ returns the current setting of NUMERIC FUZZ.

The following is an example:

```
FUZZ()    ->    0     /* by default */
```

INSERT

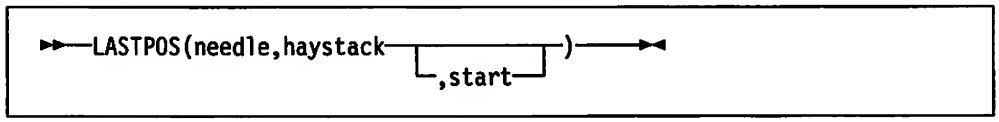


INSERT inserts the string *new*, padded to length *length*, into the string *target* after the *n*th character. If specified, *n* must be a non-negative whole number. If *n* is greater than the length of the target string, padding is added there also. The default *pad* character is a blank. The default value for *n* is 0, which means insert before the beginning of the string.

The following are some examples:

```
INSERT(' ','abcdef',3)          ->    'abc def'
INSERT('123','abc',5,6)         ->    'abc  123   '
INSERT('123','abc',5,6,'+')     ->    'abc++123+++'
INSERT('123','abc')             ->    '123abc'
INSERT('123','abc',,5,'-')      ->    '123--abc'
```

LASTPOS (Last Position)

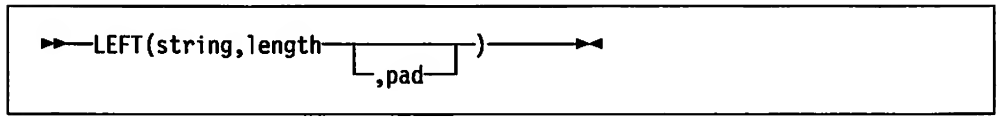


LASTPOS returns the position of the last occurrence of one string, *needle*, in another, *haystack* (see "POS (Position)" on page 94); 0 is returned if *needle* is the null string or is not found. By default, the search starts at the last character of *haystack* (that is, *start*=*LENGTH(haystack)*) and scans backwards. You can override this by specifying *start* as the point at which the backwards scan starts. The *start* parameter must be a positive whole number; it defaults to *LENGTH(haystack)* if larger than that value or omitted.

The following are some examples:

```
LASTPOS(' ','abc def ghi')      ->    8
LASTPOS(' ','abcdefghi')        ->    0
LASTPOS('xy','efgxyz')          ->    4
LASTPOS(' ','abc def ghi',7)    ->    4
```

LEFT

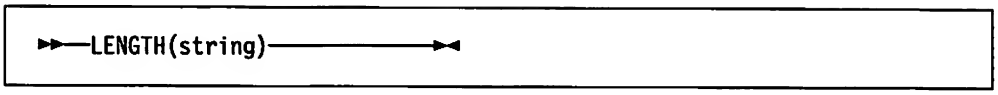


LEFT returns a string of length *length*, containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be non-negative. The LEFT function is exactly equivalent to *SUBSTR(string,1,length[,pad])*.

The following are some examples:

```
LEFT('abc d',8)        ->    'abc d   '
LEFT('abc d',8,'.')    ->    'abc d...'
LEFT('abc  def',7)     ->    'abc  de'
```

LENGTH

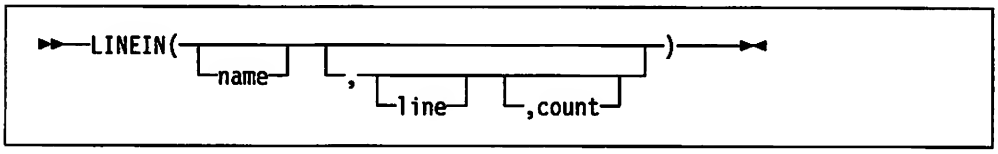


LENGTH returns the length of `string`.

The following are some examples:

```
LENGTH('abcdefgh')    ->    8
LENGTH('abc defg')    ->    8
LENGTH('')            ->    0
```

LINEIN (Lines of Input to Read)



LINEIN returns *count* (0 or 1) lines read from the character input stream *name*. (See Chapter 8, "Input and Output Streams," for a discussion of the REXX input/output model.) The form of the *name* is implementation-dependent. If you omit *name*, the line is read from the default input stream, STDIN:, in the OS/2 program. The default *count* is 1.

For persistent streams, a read position is maintained for each stream. In the OS/2 implementation, this is the same as the write position. Any read from the stream starts at the current read position by default. A call to LINEIN will return a partial line if the current read position is not at the start of a line. When the read is completed, the read position is moved to the beginning of the next line. The read position may be set to the beginning of the stream by giving *line* a value of 1—the only valid value for *line* in the OS/2 environment.

If you give a *count* of 0, then no characters are read and the null string is returned.

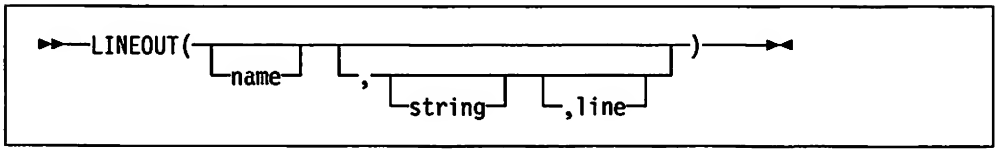
For transient streams, if a complete line is not available in the stream, then execution of the program normally stops until the line is complete. If, however, it is impossible for a line to be completed due to an error or other problem, the NOTREADY condition is raised (see "Errors During Input and Output" on page 147) and LINEIN returns whatever characters are available.

The following are some examples:

```
LINEIN()                                /* Reads one line from the    */
                                        /* default input stream;       */
                                        /* normally this is an entry  */
                                        /* typed at the keyboard      */

myfile = 'ANYFILE.TXT'
LINEIN(myfile)      -> 'Current line'   /* Reads one line from        */
                                        /* ANYFILE.TXT, beginning     */
                                        /* at the current read        */
                                        /* position. (If first call,  */
                                        /* file is opened and the     */
                                        /* first line is read.)       */

LINEIN(myfile,1,1) ->  'first line'     /* Opens and reads the first  */
                                        /* line of ANYFILE.TXT (if    */
                                        /* the file is already open,  */
                                        /* reads first line); sets    */
                                        /* read position on the       */
                                        /* second line.               */

LINEIN(myfile,1,0) ->  ''               /* No read; opens ANYFILE.TXT */
                                        /* (if file is already open,  */
                                        /* sets the read position to  */
                                        /* the first line).           */

LINEIN(myfile,,0)  ->  ''               /* No read; opens ANYFILE.TXT */
                                        /* (no action if the file is  */
                                        /* already open).             */

LINEIN("QUEUE:")   ->  'Queue line'     /* Read a line from the queue; */
                                        /* If the queue is empty, the  */
                                        /* program waits until a line  */
                                        /* is put on the queue.        */
```

Note: If the intention is to read complete lines from the default character stream, as in a simple dialog with a user, use the PULL or PARSE PULL instructions instead for simplicity and for improved programmability. The PARSE LINEIN instruction is also useful in certain cases.

LINEOUT (Lines of Output to Write)



LINEOUT returns the count of lines remaining after attempting to write *string* to the character output stream *name*. (See Chapter 8, "Input and Output Streams," for a discussion of the REXX input/output model.) The count is either 0 (meaning the line was successfully written) or 1 (meaning that an error occurred while writing the line). *string* can be the null string, in which case only the action associated with completing a line is taken. LINEOUT adds a line-feed and a carriage-return character to the end of *string*.

The form of the *name* is implementation dependent. If you omit *name*, the line is written to the default output stream, STDOUT: (normally the display), in the OS/2 program.

For persistent streams, a write position is maintained for each stream. In the OS/2 program implementation, this is the same as the read position. Any write to the stream starts at the current write position by default. Characters written by a call to LINEOUT may be added to a partial line. LINEOUT conceptually terminates a line at the *end* of each call. When the write is completed, the write position is set to the beginning of the line following the one written. The initial write position is the end of the stream, so that calls to LINEOUT normally append lines to the end of the stream.

You can set the write position to the first character of a persistent stream by giving a value of 1 (the only valid value) for *line*.

Note: In some environments, overwriting a stream using CHAROUT or LINEOUT can erase (destroy) all existing data in the stream. This is not, however, the case in the OS/2 environment.

You can omit the *string* for persistent streams. If you specify *line*, the write position is set to the beginning of the stream, but nothing is written to the stream, and 0 is returned. If you specify neither *line* nor *string*, the write position is set to the end of the stream. This use of LINEOUT has the effect of closing the stream in environments (such as the OS/2 environment) that support this concept.

Execution of the program normally stops until the output operation is effectively completed. If, however, it is impossible for a line to be written, the NOTREADY condition is raised (see "Errors During Input and Output" on page 147) and LINEOUT returns with a result of 1 (that is, the residual count of lines written).

The following are some examples:

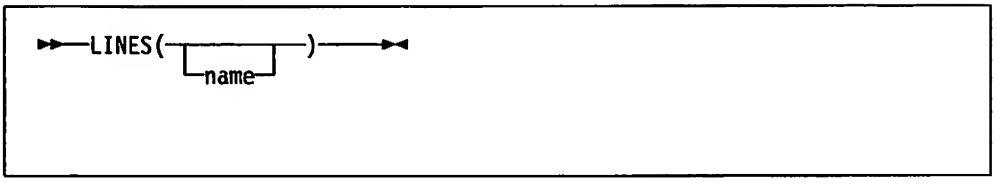
```
LINEOUT(,'Display this')          /* Writes string to the default   */
                                  /* output stream (normally, the   */
                                  /* display); returns 0 if         */
                                  /* successful                     */

myfile = 'ANYFILE.TXT'
LINEOUT(myfile,'A new line')      /* Opens the file ANYFILE.TXT and */
                                  /* appends the string to the end. */
                                  /* If the file is already open,   */
                                  /* the string is written at the   */
                                  /* current write position.        */
                                  /* Returns 0 if successful.       */

LINEOUT(myfile,'A new start',1)   /* Opens the file (if not already */
                                  /* open); overwrites first line   */
                                  /* with a new line.               */
                                  /* Returns 0 if successful.       */

LINEOUT(myfile,,1)                /* Opens the file (if not already */
                                  /* open). No write; sets write    */
                                  /* position at first character.   */

LINEOUT(myfile)                   /* Closes ANYFILE.TXT             */
```

LINEOUT is often most useful when called as a subroutine. The return value is then available in the variable RESULT. For example:

```
Call LINEOUT 'A:rexx.bat','Shell',1
Call LINEOUT ,'Hello'
```

Note: If the lines are to be written to the default output stream without the possibility of error, use the SAY instruction instead.

LINES (Lines Remaining to Read)



LINES returns 1 if any data remains between the current read position and the end of the character input stream *name*; returns 0 if no data remains. In effect, LINES reports whether a read action that CHARIN (see page 76) or LINEIN (see page 89) performs will succeed. (See page Chapter 8, "Input and Output Streams," for a discussion of the REXX input/output model.)

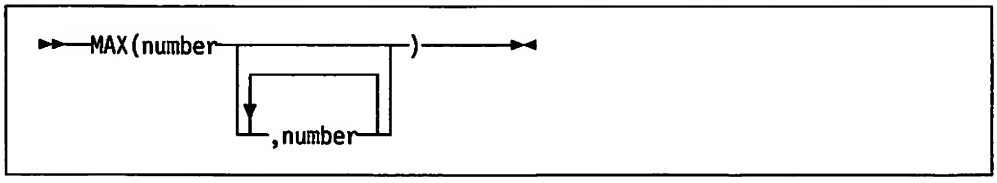
The form of the *name* is implementation-dependent. If you omit *name*, then the presence or absence of data in the default input stream (STDIN:) is returned. For OS/2 devices, LINES always returns 1. For QUEUE, the actual number of lines is returned.

The following are some examples:

```
LINES(myfile)     ->    0    /* at end of the file   */

LINES()           ->    1    /* data remains in the  */
                             /* default input stream */
                             /* STDIN:         */

LINES("COM1:")    ->    1    /* An OS/2 device name  */
                             /* always returns '1'   */
```

Note: The CHARS function returns the number of characters in a persistent stream or the presence of data in a transient stream.

MAX (Maximum)

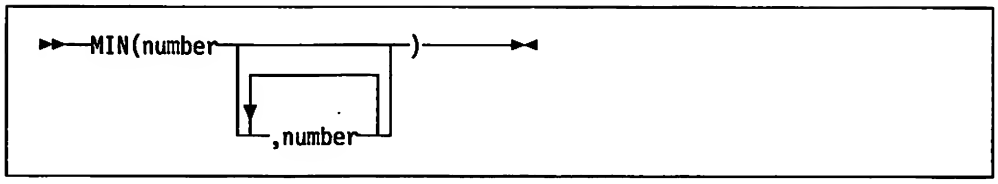


MAX returns the largest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. You can specify up to 20 numbers, and can nest calls to MAX if more arguments are needed.

The following are some examples:

```
MAX(12,6,7,9)                                                  ->   12
MAX(17.3,19,17.03)                                             ->   19
MAX(-7,-3,-4.3)                                                ->   -3
MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MAX(21,22))  ->   22
```

MIN (Minimum)

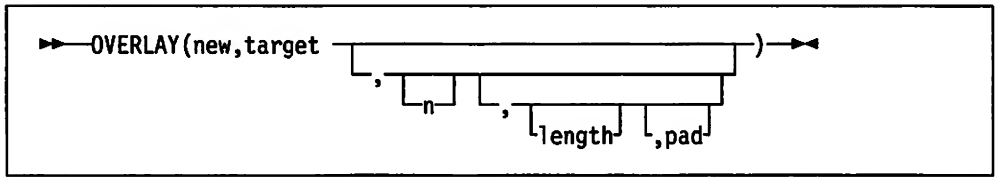


MIN returns the smallest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to 20 numbers can be specified, although calls to MIN can be nested if more arguments are needed.

The following are some examples:

```
MIN(12,6,7,9)          ->    6
MIN(17.3,19,17.03)     ->    17.03
MIN(-7,-3,-4.3)        ->    -7
```

OVERLAY

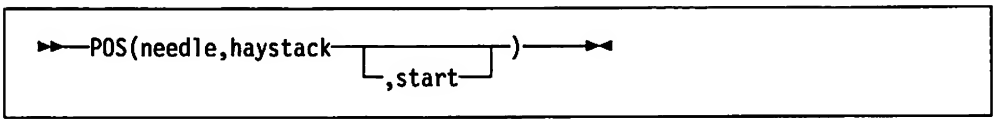


OVERLAY returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. If you specify *length*, it must be positive or zero. If *n* is greater than the length of the target string, padding is added before the *new* string. The default *pad* character is a blank and the default value for *n* is 1. If you specify *n*, it must be a positive whole number.

The following are some examples:

```
OVERLAY(' ','abcdef',3)          ->   'ab def'
OVERLAY('.','abcdef',3,2)        ->   'ab. ef'
OVERLAY('qq','abcd')             ->   'qqcd'
OVERLAY('qq','abcd',4)           ->   'abcqq'
OVERLAY('123','abc',5,6,'+')     ->   'abc+123+++'
```

POS (Position)

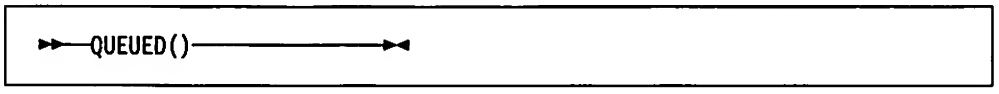


POS returns the position of one string, *needle*, in another, *haystack*. Returns 0 if *needle* is not found. By default, the search starts at the first character of *haystack* (that is, the value of *start* is 1). You can override this by specifying *start* (which must be a positive whole number) as the point at which the search starts.

The following are some examples:

```
POS('day','Saturday')       ->   6
POS('x','abc def ghi')      ->   6
POS(' ','abc def ghi')      ->   4
POS(' ','abc def ghi',5)    ->   8
```

QUEUED

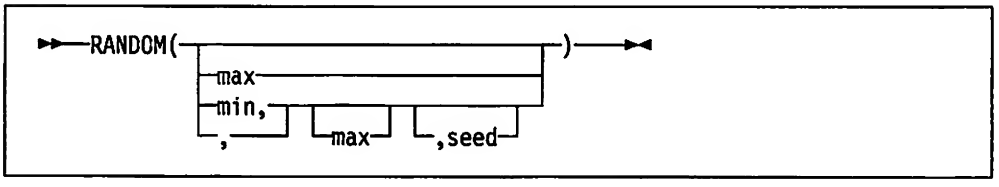


QUEUED returns the number of lines remaining in the currently active REXX data queue at the time when the function is invoked.

The following is an example:

```
QUEUED()    ->    5    /* Perhaps */
```

RANDOM



RANDOM returns a quasi-random, non-negative whole number in the range *min* to *max* inclusively. If you specify *max* or *min* or both, *max* minus *min* cannot exceed 100000. *min* and *max* default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific *seed* as the third argument. This *seed* must be a whole number.

The following are some examples:

```
RANDOM()          ->    305
RANDOM(5,8)       ->      7
RANDOM(2)         ->      0  /*  0  to  2    */
RANDOM(2,)        ->    747  /*  2  to  999  */
RANDOM(,,1983)    ->    123  /* reproducible */
```

Notes:

1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a *seed* only the first time. For example, to simulate 40 throws of a six-sided, unbiased die, use:

```
sequence = RANDOM(1,6,12345)  /* any number would */
                              /* do for a seed    */
do 39
   sequence = sequence RANDOM(1,6)
   end
say sequence
```

   The numbers are generated mathematically, using the initial *seed*, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial *seed* almost certainly produces a different sequence.

2. The random number generator is global for an entire program; the current seed is not saved across internal routine calls.

3. The actual random number generator used may differ from implementation to implementation.

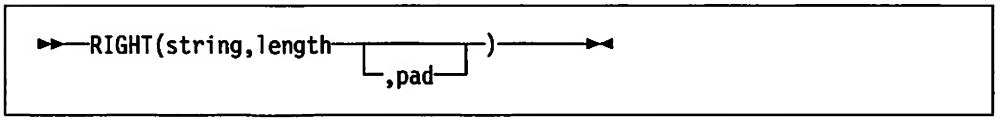
REVERSE

REVERSE returns *string*, swapped end for end.

The following are some examples:

```
REVERSE('ABc.')    ->    '.cBA'
REVERSE('XYZ ')    ->    ' ZYX'
```

RIGHT



RIGHT returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. *length* must be non-negative.

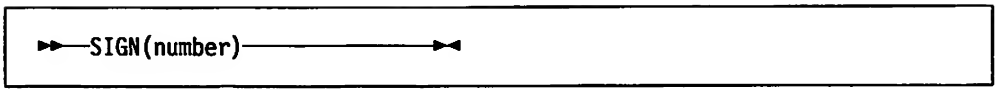
The following are some examples:

```
RIGHT('abc  d',8)     ->    '  abc  d'
RIGHT('abc def',5)    ->    'c def'
RIGHT('12',5,'0')     ->    '00012'
```

SETLOCAL

This is an OS/2-specific function. See page 113.

SIGN

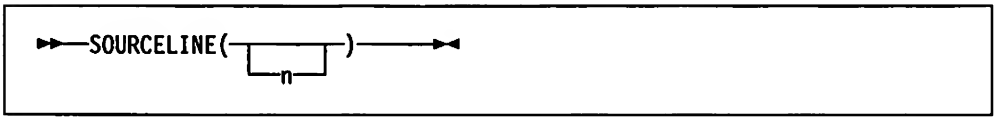


SIGN returns a number that indicates the sign of *number*. *number* is first rounded according to standard REXX rules, just as though the operation *number*+*0* had been carried out. Returns −1 if number is less than 0, returns 0 if it is 0, and returns 1 if it is greater than 0.

The following are some examples:

```
SIGN('12.3')       ->    1
SIGN(' -0.307')    ->   -1
SIGN(0.0)          ->    0
```

SOURCELINE

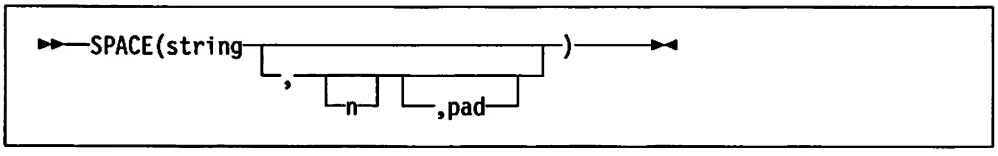


SOURCELINE returns the line number of the final line in the source file, if you omit *n*, or returns the *n* the line in the source file if you specify *n*. If specified, *n* must be a positive whole number and must not exceed the number that a call to SOURCELINE with no arguments returns.

The following are some examples:

```
SOURCELINE()    ->  10
SOURCELINE(1)   ->  '/* This is a 10-line program */'
```

SPACE

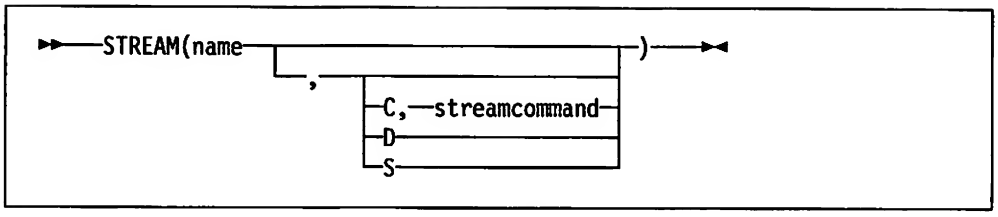


SPACE formats the blank-delimited words in *string* with *n* *pad* characters between each word. The *n* must be non-negative. If it is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is 1 and the default *pad* character is a blank.

The following are some examples:

```
SPACE('abc  def  ')          ->    'abc def'
SPACE('  abc def',3)         ->    'abc   def'
SPACE('abc  def  ',1)        ->    'abc def'
SPACE('abc  def  ',0)        ->    'abcdef'
SPACE('abc  def  ',2,'+')    ->    'abc++def'
```

STREAM



STREAM returns a string describing the state of, or the result of an operation upon, the character stream *name*. (See Chapter 8, "Input and Output Streams," for a discussion of the REXX input/output model.) This function is used to request information on the state of an input or output stream or to carry out some specific operation on the stream.

The first argument, *name*, specifies the stream to be accessed. The second argument can be one of the following strings (of which only the first letter is needed) describing the action to be carried out:

Command An operation (specified by the *streamcommand* given as the third argument) to be applied to the selected input or output stream. The string that is returned depends on the command performed and may be the null string. The possible input strings for the *streamcommand* argument are described in the following text.

Description Also returns the current state of the specified stream. It is identical to the State operation, except that the returned string is followed by a colon and, if available, additional information about the ERROR or NOTREADY states.

State Returns a string that indicates the current state of the specified stream. This is the default operation.

When used with the State option, STREAM returns one of the following strings:

'ERROR' The stream has been subject to an erroneous operation (possibly during input, output, or through the STREAM function—see "Errors During Input and Output" on page 147). You may be able to obtain additional information about the error by invoking the STREAM function with a request for the implementation-dependent description.

'NOTREADY' The stream is known to be in a state such that normal input or output operations attempted upon it would raise the NOTREADY condition. (See page 147). For example, a simple input stream may have a defined length; an attempt to read that stream (with the CHARIN or LINEIN built-in functions) beyond that limit may make the stream unavailable until the stream has been closed (for example, with LINEOUT(*name*)) and then reopened.

'READY' The stream is known to be in a state such that normal input or output operations can be attempted (this is the usual state for a stream, though it does not guarantee that any particular operation will succeed).

'UNKNOWN' The state of the stream is unknown. In OS/2 implementations, this generally means that the stream is closed (or has not yet been opened). However, this response can be used in other environments to indicate that the state of the stream cannot be determined.

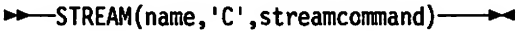
Note: The state (and operation) of an input or output stream is global to a REXX program, in that it is not saved and restored across function and subroutine calls (including those a CALL ON condition trap causes).

Stream Commands

The following stream commands can be used to:

- Open a stream for reading or writing
- Close a stream at the end of an operation
- Position the read or write position within a persistent stream (for example, a file)
- Get information about a stream (its existence, size, and last edit date).

The *streamcommand* argument must be used when—and only when—you select the operation C (command). The syntax is:



In this form, the STREAM function itself returns a string corresponding to the given *streamcommand* if the command is successful. If the command is unsuccessful, STREAM returns an error message string in the same form as that supplied by the D (Description) operation.

Command strings: The argument *streamcommand* can be any expression that REXX evaluates as one of the following command strings:

'OPEN' Opens the named stream. The default for **'OPEN'** is to open the stream for both reading and writing data. To specify whether *name* is only to be read or only to be written to, add the word READ or WRITE to the command string.

The STREAM function itself returns 'READY' if the named stream is successfully opened or an appropriate error message if unsuccessful.

Examples:

```
stream(strout,'c','open')
stream(strout,'c','open write')
stream(strinp,'c','open read')
```

'CLOSE' Closes the named stream. The STREAM function itself returns 'READY' if the named stream is successfully closed or an appropriate error message otherwise. If an attempt is made to close an unopened file, then STREAM() returns a null string ("").

Example:

```
stream('STRM.TXT','c','close')
```

'SEEK *offset* **'** Sets the read or write position a given number (*offset*) within a persistent stream.

Note: In the OS/2 program, the read and write positions are the same. (See page 141 for a discussion of read and write positions in a persistent stream.) To use this command, the named stream must first be opened (with the **'OPEN'** stream command, described previously).

The *offset* number can be preceded by one of the following characters:

= Explicitly specifies the *offset* from the beginning of the stream. This is the default if no prefix is supplied.

< Specifies *offset* from the end of the stream.

+ Specifies *offset* forward from the current read or write position.

- Specifies *offset* backward from the current read or write position.

The STREAM function itself returns the new position in the stream if the read or write position is successfully located; an appropriate error message is displayed otherwise.

Examples:

```
stream(name,'c','seek =2')
stream(name,'c','seek +15')
stream(name,'c','seek -7')
fromend  = 125
stream(name,'c','seek <'fromend)
```

Used with these stream commands, the STREAM function returns the following specific information about a stream:

'QUERY EXISTS' Returns the full-path specification of the named stream, if it exists or a null string if otherwise. For example:

```
stream('..\file.txt','c','query exists')
```

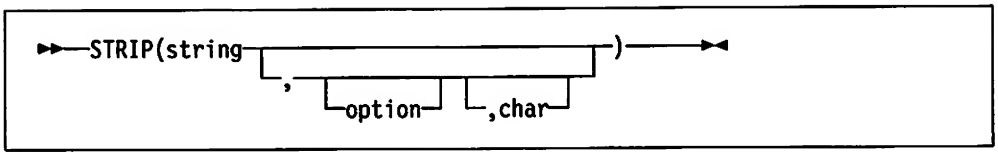
'QUERY SIZE' Return the size in bytes of a persistent stream. For example:

```
stream('..\file.txt','c','query size')
```

'QUERY DATETIME' Returns the date and time stamps of a stream. For example:

```
stream('..\file.txt','c','query datetime')
```

STRIP



STRIP returns *string* with leading and trailing characters removed, based on the *option* you specify. Valid options (of which only the capitalized letter is significant, all others are ignored) are:

Both Removes both leading and trailing characters from *string*. This is the default.

Leading Removes leading characters from *string*.

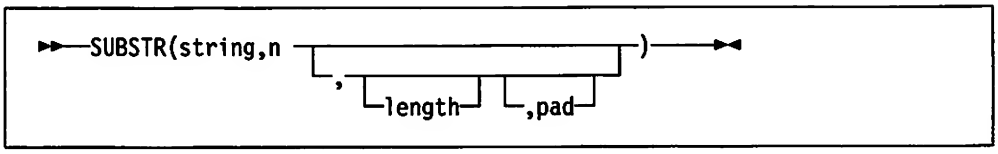
Trailing Removes trailing characters from *string*.

The third argument, *char*, specifies the character to remove; the default is a blank. If you specify *char*, it must be exactly 1 character long.

The following are some examples:

```
STRIP('  ab c  ')        ->   'ab c'
STRIP('  ab c  ','L')    ->   'ab c  '
STRIP('  ab c  ','t')    ->   '  ab c'
STRIP('12.7000',,0)      ->   '12.7'
STRIP('0012.700',,0)     ->   '12.7'
```

SUBSTR (Substring)



SUBSTR returns the substring of *string* that begins at the *n*th character and is of length *length*, padded with *pad* if necessary. *n* must be a positive whole number.

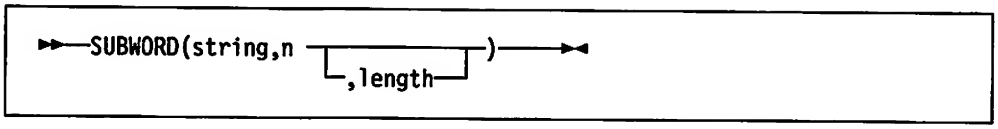
If you omit *length*, the rest of the string is returned. The default *pad* character is a blank.

The following are some examples:

```
SUBSTR('abc',2)          ->   'bc'
SUBSTR('abc',2,4)        ->   'bc  '
SUBSTR('abc',2,6,'.')    ->   'bc....'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, especially if more than one substring is to be extracted from a string.

SUBWORD

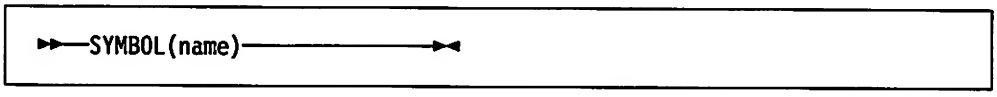


SUBWORD returns the substring of *string* that starts at the *n*th word, and is of length *length*, blank-delimited words. *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes all blanks between the selected words.

The following are some examples:

```
SUBWORD('Now is the  time',2,2)    ->    'is the'
SUBWORD('Now is the  time',3)      ->    'the  time'
SUBWORD('Now is the  time',5)      ->    ''
```

SYMBOL



SYMBOL returns the state of the symbol named by *name*. SYMBOL returns BAD if *name* is not a valid REXX symbol. SYMBOL returns VAR if it is the name of a variable (that is, a symbol that has been assigned a value). Otherwise, SYMBOL returns LIT, indicating that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a literal).

As with symbols in REXX expressions, lowercase characters in *name* are translated to uppercase and substitution in a compound name occurs if possible.

Note: You should specify *name* as a literal string (or derived from an expression) to prevent substitution before it is passed to the function.

The following are some examples:

```
/* following: Drop A.3;  J=3 */
SYMBOL('J')       ->   'VAR'
SYMBOL(J)         ->   'LIT' /* has tested "3"     */
SYMBOL('a.j')     ->   'LIT' /* has tested "A.3"   */
SYMBOL(2)         ->   'LIT' /* a constant symbol  */
SYMBOL('*')       ->   'BAD' /* not a valid symbol */
```

TIME

TIME returns the local time in the 24-hour clock format hh:mm:ss (hours, minutes, and seconds) by default; for example, 04:41:37.

You can use the following options (for which only the capitalized letter is needed) to obtain alternative formats, or to gain access to the elapsed-time clock:

Civil Returns hh:mmxx, the time in Civil format, in which the hours may take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters am or pm to distinguish times in the morning (midnight 12:00am through 11:59am) from noon and afternoon (noon 12:00pm through 11:59pm). The hour has no leading zero. The minute field shows the current minute, rather than the nearest minute, for consistency with other TIME results.

Elapsed Returns ssssssss.uu0000, the number of seconds.hundredths since the elapsed-time clock was started or **Reset** (see the following text). The returned number has no leading zeros, but always has four trailing zeros in the decimal portion. It is not affected by the setting of NUMERIC DIGITS.

Hours Returns number of hours since midnight in the format hh (no leading zeros).

Long Returns time in the format hh:mm:ss.uu0000 (where uu is the fraction of seconds in hundredths of a second.

Minutes Returns number of minutes since midnight in the format mmmm (no leading zeros).

Normal Returns the time in the default format hh:mm:ss, as described previously.

Reset Returns ssssssss.uu0000, the number of seconds.hundredths since the elapsed-time clock was started or reset and also resets the elapsed-time clock to zero. The returned number has no leading zeros, but always has four trailing zeros in the decimal portion.

Seconds Returns number of seconds since midnight in the format ssss (no leading zeros).

The following are some examples:

```
TIME('L')    ->   '16:54:22.120000'  /* Perhaps */
TIME()       ->   '16:54:22'
TIME('H')    ->   '16'
TIME('M')    ->   '1014'           /* 54 + 60*16 */
TIME('S')    ->   '60862'  /* 22 + 60*(54+60*16) */
TIME('N')    ->   '16:54:22'
TIME('C')    ->   '4:54pm'
```

The elapsed-time clock:

The elapsed-time clock may be used for measuring real time intervals. On the first call to the elapsed-time clock, the clock is started, and both `TIME('E')` and `TIME('R')` return `0`.

The clock is saved across internal routine calls, which is to say that an internal routine inherits the time clock its caller started. Any timing the caller is doing is not affected, even if an internal routine resets the clock.

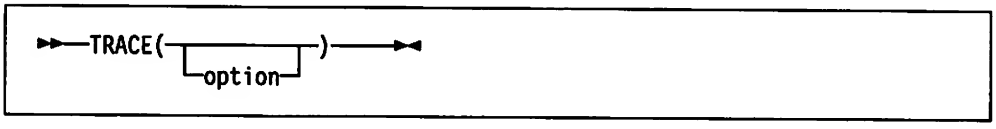
The following is an example of the elapsed-time clock:

```
time('E')    ->    0          /* The first call */
/* pause of one second here */
time('E')    ->    1.020000   /* or thereabouts */
/* pause of one second here */
time('R')    ->    2.030000   /* or thereabouts */
/* pause of one second here */
time('R')    ->    1.050000   /* or thereabouts */
```

Note: See "DATE" on page 82 about consistency of times within a single expression. The elapsed-time clock is synchronized to the other calls to TIME and DATE, so multiple calls to the elapsed-time clock in a single expression always return the same result. For the same reason, the interval between two normal TIME and DATE results can be calculated exactly using the elapsed-time clock.

Implementation maximum: Should the number of seconds in the elapsed time exceed nine digits (equivalent to over 31.6 years), an error will result.

TRACE



TRACE returns trace actions currently in effect.

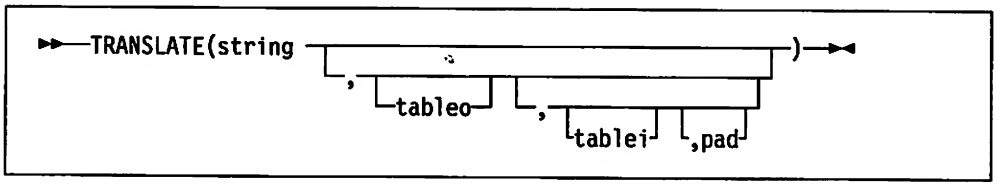
If *option* is supplied, it must be the valid prefix (?), one of the alphabetic character options (that is, starting with A, C, E, F, I, L, N, O, or R) associated with the TRACE instruction, or both. (See "TRACE" on page 60 for further details.) The function uses *option* to alter the effective trace action (such as tracing labels). Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debug is active.

Unlike the TRACE instruction, *option* cannot be a number.

The following are some examples:

```
TRACE()       ->   '?R' /* maybe */
TRACE('O')    ->   '?R' /* also sets tracing off    */
TRACE('?I')   ->   'O'  /* now in interactive debug */
```

TRANSLATE



TRANSLATE returns *string* with each character translated to another character or unchanged. You can also use this function to reorder the characters in *string*. The output table is *tableo*, the input translate table is *tablei*, the default is XRANGE('00'x,'FF'x). The default *pad* is a blank. The tables can be of any length; the first occurrence of a character in the input table is the one that is used if there

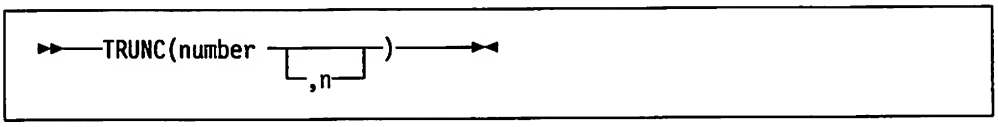
are duplicates. If you specify neither translate table, *string* is translated to uppercase (for example a lowercase a-z becomes an uppercase A-Z). The output table defaults to the null string and is padded with *pad* or truncated as necessary.

The following are some examples:

```
TRANSLATE('abcdef')                    ->   'ABCDEF'
TRANSLATE('abbc','&','b')              ->   'a&&c'
TRANSLATE('abcdef','12','ec')          ->   'ab2d1f'
TRANSLATE('abcdef','12','abcd','.')    ->   '12..ef'
TRANSLATE('4123','abcd','1234')        ->   'dabc'
```

Note: The last example shows how to use the TRANSLATE function to reorder the characters in a string. In the example, the last character of any four-character string specified as the second argument would be moved to the beginning of the string.

TRUNC



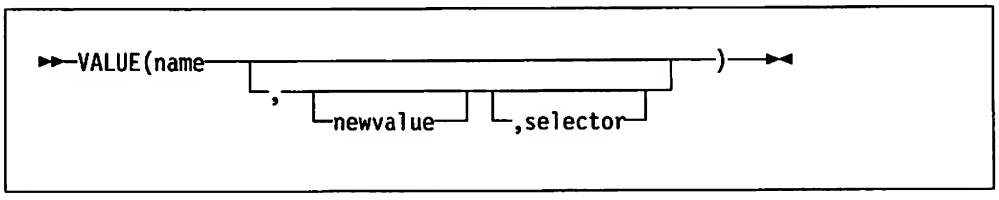
TRUNC returns the integer part of *number* and *n* decimal places. The default *n* is zero and returns an integer with no decimal point. If you specify *n*, it must be a non-negative whole number. *number* is first rounded according to standard REXX rules, as though the operation *number+0* had been carried out. The number is then truncated to *n* decimal places (or trailing zeros are added if needed to make up the specified length). The result is never in exponential form.

The following are some examples:

```
TRUNC(12.3)           ->   12
TRUNC(127.09782,3)    ->   127.097
TRUNC(127.1,3)        ->   127.100
TRUNC(127,2)          ->   127.00
```

Note: The *number* parameter is rounded according to the current setting of NUMERIC DIGITS if necessary before the function processes it.

VALUE



VALUE returns the value of the symbol that *name* represents and optionally assigns it a new value. By default, VALUE refers to the current REXX-variables environment, but you can select other external collections of variables. If you use the function to refer to REXX variables, then *name* must be a valid REXX symbol. (You can confirm this by using the SYMBOL function.) Lowercase characters in *name* are translated to uppercase. If *name* is a compound symbol, then REXX

substitutes symbol values to produce the derived name of the symbol (see "Compound Symbols" on page 19).

If you specify *newvalue*, then the named variable is assigned this new value. This does not affect the result returned; that is, the function returns the value of *name* as it was before the new assignment.

The following are some examples:

```
/* After: Drop A3; A33=7; K=3; fred='K'; list.5='Hi' */
VALUE('a'k)     -> 'A3'
VALUE('a'k||k)  -> '7'
VALUE('fred')   -> 'K'  /* looks up FRED   */
VALUE(fred)     -> '3'  /* looks up K      */
VALUE(fred,5)   -> '3'  /* and sets K=5    */
VALUE(fred)     -> '5'
VALUE('LIST.'k) -> 'Hi' /* looks up LIST.5 */
```

To use VALUE to manipulate the OS/2 environment variables, *selector* must be the string 'OS2ENVIRONMENT' or an expression so evaluated. In this case, the variable *name* need not be a valid REXX symbol. When VALUE is used to set or change the value of an environment variable, the new value is retained after the REXX procedure ends.

The following are some examples:

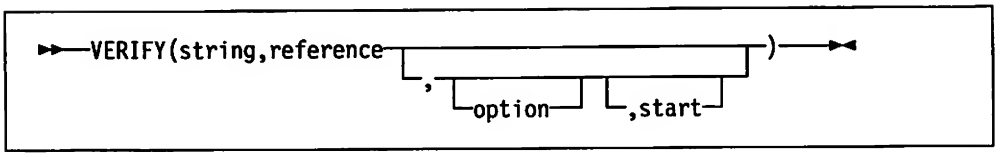
```
/* Given that an external variable FRED has a value of 4      */
share = 'OS2ENVIRONMENT'
say VALUE('fred',7,share)       /* says '4' and assigns       */
                                /* FRED a new value of 7      */

say VALUE('fred',,share)        /* says '7'                   */

/* After this procedure ends, FRED again has a value of 4     */

/* Accessing and changing OS/2 environment entries            */
env = 'OS2ENVIRONMENT'
new = 'C:\LIST\PROD;'
say value('prompt',,env)    /* says '$i[p]' (perhaps)         */
say value('path',new,env)   /* says 'C:\EDIT\DOCS;' (perhaps) */
                            /* and sets PATH = 'C:LIST\PROD'  */

say value('path',,env)      /* says 'C:LIST\PROD'             */

/* When this procedure ends, PATH = 'C:\LIST\PROD'           */
```

Notes:

1. If the VALUE function refers to an uninitialized REXX variable, then the default value of the variable is always returned; the NOVALUE condition is not raised. A reference to an external collection of variables never raises NOVALUE.
2. The VALUE function is used when a variable contains the name of another variable, or when a name is constructed dynamically. If you specify the *name* as a single literal string, the symbol is a constant and so the string between the quotes can usually replace the whole function call. For example, `fred=VALUE('k');` is identical to the assignment `fred=k;`, unless the NOVALUE condition is being trapped. (See Chapter 7, "Conditions and Condition Traps.")

3. To effect *temporary* changes to environment variables, use the SETLOCAL and ENDLOCAL functions.

VERIFY



VERIFY returns a number that, by default, indicates whether *string* is composed only of characters from *reference*; returns `0` if all characters in *string* are in *reference*, or returns the position of the first character in *string* that is not in *reference*.

The third argument, *option*, can be any expression that results in a string starting with `N` or `M` that represents either `Nomatch` (the default) or `Match`. Only the first character of *option* is significant and it can be in uppercase or lowercase, as usual. If you specify `Match`, VERIFY returns the position of the first character in *string* that is in *reference*, or returns `0` if none of the characters are found.

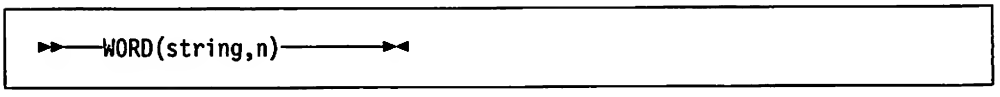
The default for *start* is `1`; thus, the search starts at the first character of *string*. You can override this by specifying a different *start* point, which must be a positive whole number.

VERIFY always returns `0` if *string* is null or if *start* is greater than `LENGTH(string)`. If *reference* is null, VERIFY returns `0` if you specify `Match`; otherwise, `1` is returned.

The following are some examples:

```
VERIFY('123','1234567890')              ->   0
VERIFY('1Z3','1234567890')              ->   2
VERIFY('AB4T','1234567890')             ->   1
VERIFY('AB4T','1234567890','M')         ->   3
VERIFY('AB4T','1234567890','N')         ->   1
VERIFY('1P3Q4','1234567890',,3)         ->   4
VERIFY('AB3CD5','1234567890','M',4)     ->   6
```

WORD



WORD returns the *n*th blank-delimited word in *string* or returns the null string if fewer than *n* words are in *string*; *n* must be a positive whole number. This function is exactly equivalent to `SUBWORD(string,n,1)`.

The following are some examples:

```
WORD('Now is the time',3)    ->    'the'
WORD('Now is the time',5)    ->    ''
```

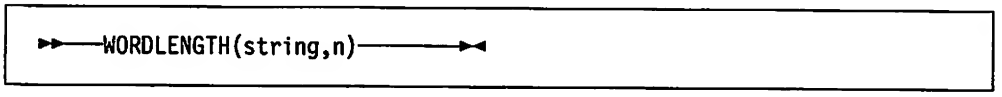
WORDINDEX

WORDINDEX returns the position of the first character in the *n*th blank-delimited word in *string* or returns 0 if fewer than *n* words are in *string*; *n* must be a positive whole number.

The following are some examples:

```
WORDINDEX('Now is the time',3)    ->    8
WORDINDEX('Now is the time',6)    ->    0
```

WORDLENGTH

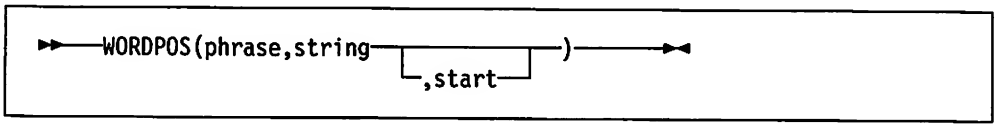


WORDLENGTH returns the length of the *n*th blank-delimited word in *string* or returns 0 if fewer than *n* words are in *string*; *n* must be a positive whole number.

The following are some examples:

```
WORDLENGTH('Now is the time',2)       ->    2
WORDLENGTH('Now comes the time',2)    ->    5
WORDLENGTH('Now is the time',6)       ->    0
```

WORDPOS (Word Position)



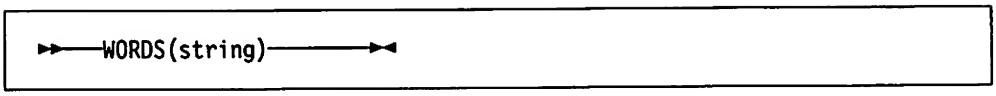
WORDPOS returns the word number of the first word of *phrase* found in *string* or returns 0 if *phrase* is not found. Multiple blanks between words in either *phrase* or *string* are treated as a single blank for the comparison; otherwise, the words must match exactly.

By default the search starts at the first word in *string*. You can override this by specifying *start*, which must be positive, the word at which to start the search.

The following are some examples:

```
WORDPOS('the','now is the time')              -> 3
WORDPOS('The','now is the time')              -> 0
WORDPOS('is the','now is the time')           -> 2
WORDPOS('is   the','now is the time')         -> 2
WORDPOS('is   time ','now is   the time')     -> 0
WORDPOS('be','To be or not to be')            -> 2
WORDPOS('be','To be or not to be',3)          -> 6
```

WORDS

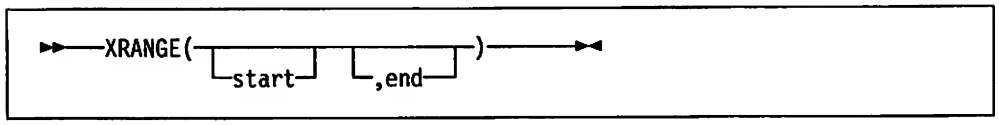


WORDS returns the number of blank-delimited words in *string*.

The following are some examples:

```
WORDS('Now is the time')    ->    4
WORDS(' ')                  ->    0
```

XRANGE

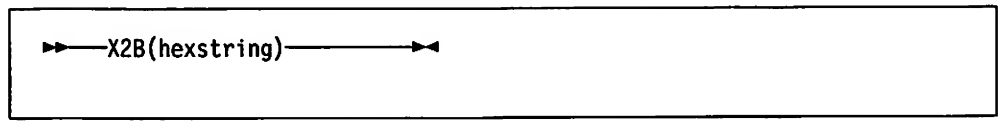


XRANGE returns a string of all 1-byte codes between and including the values *start* and *end*. The default value for *start* is '00'x, and the default value for *end* is 'FF'x. If *start* is greater than *end*, the values wrap from 'FF'x to '00'x. If specified, *start* and *end* must be single characters.

The following are some examples:

```
XRANGE('a','f')      ->   'abcdef'
XRANGE('03'x,'07'x)  ->   '0304050607'x
XRANGE(,'04'x)       ->   '0001020304'x
XRANGE('i','j')      ->   '898A8B8C8D8E8F9091'x  /* EBCDIC */
XRANGE('FE'x,'02'x)  ->   'FEFF000102'x
XRANGE('i','j')      ->   'ij'                   /* ASCII  */
```

X2B (Hexadecimal to Binary)



X2B returns a string, in character format, that represents *hexstring* converted to binary. The *hexstring* is a string of hexadecimal characters. It can be of any length. Each hexadecimal character is converted to a string of four binary digits. You can optionally add blanks to *hexstring* (at byte boundaries only, not leading or trailing positions) to aid readability; they are ignored.

The returned string has a length that is a multiple of four, and does not include any blanks.

If *hexstring* is null, X2B returns a null string.

The following are some examples:

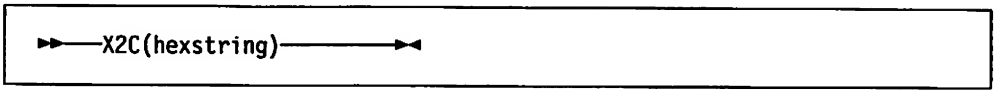
```
X2B('C3')        ==  '11000011'
X2B('7')         ==  '0111'
X2B('1 C1')      ==  '000111000001'
```

You can combine X2B() with the functions D2X() and C2X() to convert decimal numbers or character strings into binary form.

The following are some examples:

```
X2B(C2X('C3'x))  ==  '11000011'
X2B(D2X('129'))  ==  '10000001'
X2B(D2X('12'))   ==  '1100'
```

X2C (Hexadecimal to Character)



X2C returns a string, in character format, that represents *hexstring* converted to character. Since *hexstring* is a string of hexadecimal characters, in a literal string notation, the returned string is half as many bytes. *hexstring* can be of any length. You can optionally add blanks to *hexstring* (at byte boundaries only, not leading or trailing positions) to aid readability; they are ignored.

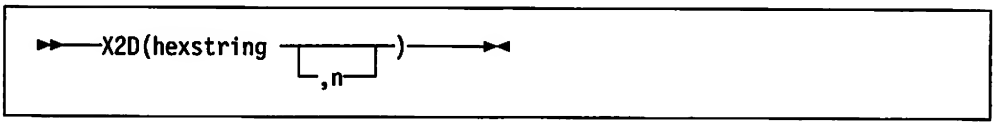
If *hexstring* is null, X2C returns a null string.

If necessary, *hexstring* is padded with a leading 0 to make an even number of hexadecimal digits.

The following are some examples:

```
X2C('4865 6c6c 6f') ->  'Hello'     /*  ASCII            */
X2C('3732 73')      ->  '72s'       /*  ASCII            */
X2C('F7F2 A2')      ->  '72s'       /*  EBCDIC                    */
X2C('F7f2a2')       ->  '72s'       /*  EBCDIC                    */
X2C('F')            ->  ' '         /*  '0F' is unprintable EBCDIC */
```

X2D (Hexadecimal to Decimal)



X2D returns the decimal representation of *hexstring*. *hexstring* is a string of hexadecimal characters. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

You can optionally add blanks to *hexstring* (at byte boundaries only, not leading or trailing positions) to aid readability; they are ignored.

If *hexstring* is null, X2D returns 0.

If you do not specify *n*, *hexstring* is processed as an unsigned binary number.

The following are some examples:

```
X2D('0E')        ->   14
X2D('81')        ->   129
X2D('F81')       ->   3969
X2D('FF81')      ->   65409
X2D('c6 f0'X)    ->   240
```

If you specify *n*, the given *hexstring* is padded on the left with zeros (note, not *sign-extended*), or truncated on the left to *n* characters. The resulting string of *n* hexadecimal digits is taken to be a signed binary number; positive if the leftmost bit is OFF, and negative, in two's complement notation, if the leftmost bit is ON. If *n* is 0, X2D returns 0.

The following are some examples:

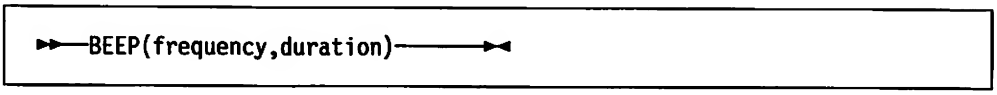
```
X2D('81',2)      ->  -127
X2D('81',4)      ->   129
X2D('F081',4)    ->   -3967
X2D('F081',3)    ->   129
X2D('F081',2)    ->   -127
X2D('F081',1)    ->   1
X2D('0031',0)    ->   0
```

OS/2-Specific Functions

These functions are specific to the OS/2 program implementation of REXX. They are not part of the SAA definition, and therefore programs that use these functions may not be supported in other SAA environments.

If you want to run a program that uses these functions on another SAA system, be sure to consult the *REXX Reference* for that system.

BEEP



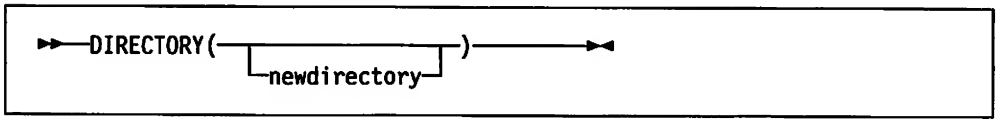
BEEP sounds the speaker at *frequency* (Hertz) for *duration* (milliseconds). The *frequency* can be any whole number in the range 37 to 32767 Hertz. The *duration* can be any number in the range 1 to 60000 milliseconds.

This routine is most useful when called as a subroutine. A null string is returned.

The following is an example:

```
/* C scale */
note.1 = 262    /* middle C */
note.2 = 294    /*    D     */
note.3 = 330    /*    E     */
note.4 = 349    /*    F     */
note.5 = 392    /*    G     */
note.6 = 440    /*    A     */
note.7 = 494    /*    B     */
note.8 = 524    /*    C     */

do i=1 to 8
  call beep note.i,250    /* hold each note for */
                          /* one-quarter second */
end
```

DIRECTORY



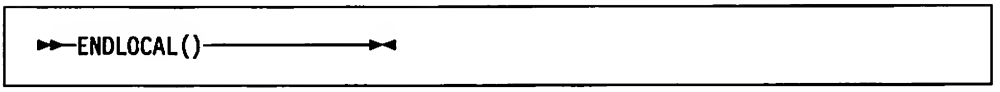
DIRECTORY returns the current directory, first changing it to *newdirectory* if an argument is supplied and the named directory exists.

The return string includes a drive letter prefix as the first two characters of the directory name. Specifying a drive letter prefix as part of *newdirectory* causes the specified drive to become the current drive. If a drive letter is not specified, then the *current drive* remains unchanged.

For example, the following program fragment saves the current directory and switches to a new directory; it performs an operation there, and then returns to the former directory.

```
/* get current directory */
curdir = directory()
/* go play a game */
newdir = directory("d:/usr/games")
if newdir = "d:/usr/games" then
   do
   fortune  /* tell a fortune */
/* return to former directory */
   call directory curdir
   end
else
   say 'Can''t find /usr/games'
```

ENDLOCAL



ENDLOCAL restores the drive directory and environment variables in effect before the last SETLOCAL function (see page 113) was executed. If ENDLOCAL is not included in a procedure, then the initial environment saved by SETLOCAL will be restored upon exiting the procedure.

ENDLOCAL returns a value of 1 if the initial environment is successfully restored and a value of 0 if no SETLOCAL has been issued or if the actions is otherwise unsuccessful.

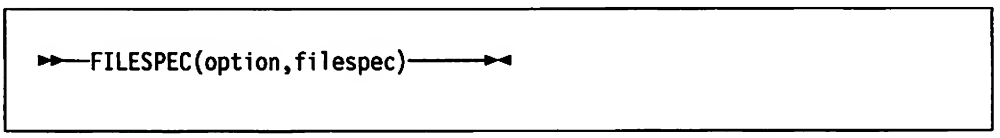
Note: Unlike their counterparts in the OS/2 batch language (the Setlocal and Endlocal statements), the REXX SETLOCAL and ENDLOCAL functions can be nested.

The following is an example:

```
n = SETLOCAL()          /* saves the current environment    */

        /* The program can now change environment   */
        /* variables (with the VALUE function) and  */
        /* then work in that changed environment.  */

n = ENDLOCAL()          /* restores the initial environment */
```

For additional examples, see "SETLOCAL" on page 113.

FILESPEC



FILESPEC returns a selected element of *filespec*, a given file specification, identified by one of the following strings for *option*:

Drive The drive letter of the given *filespec*.

Path The directory path of the given *filespec*.

Name The filename of the given *filespec*.

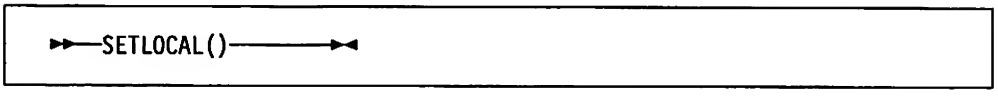
If the requested string is not found, then FILESPEC returns a null string (" ").

Note: Only the the initial letter of *option* is needed.

The following are some examples:

```
thisfile = "C:\OS2\UTIL\EXAMPLE.EXE"
say FILESPEC("drive",thisfile)     /* says "C:"          */
say FILESPEC("path",thisfile)      /* says "\OS2\UTIL\"  */
say FILESPEC("name",thisfile)      /* says "EXAMPLE.EXE" */

part = "name"
say FILESPEC(part,thisfile)        /* says "EXAMPLE.EXE" */
```

SETLOCAL



SETLOCAL saves the current working drive and directory and the current values of the OS/2 environment variables that are local to the current process.

For example, SETLOCAL can be used to save the current environment before changing selected settings with the VALUE function (see page 104). To restore the drive, directory, and environment, use the ENDLOCAL function (see page 112).

SETLOCAL returns a value of 1 if the initial drive, directory and environment are successfully saved, a value of 0 if unsuccessful. If SETLOCAL is not followed by an ENDLOCAL function in a procedure, then the initial environment saved by SETLOCAL will be restored upon exiting the procedure.

The following is an example:

```
/* current path is 'C:\PROJ\FILES' */
n = SETLOCAL()        /* saves all environment settings    */

/* Now use the VALUE function to change the PATH variable. */
p = VALUE('Path','C:\PROC\PROGRAMS'.'OS2ENVIRONMENT')

/* Programs in directory C:\PROC\PROGRAMS may now be run   */

n = ENDLOCAL()  /* restores initial environment (including */
                /* the changed PATH variable, which is     */
                /* once again 'C:\PROJ\FILES')             */
```

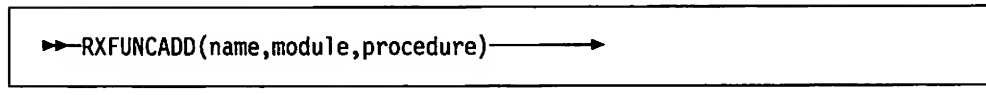
Note: Unlike their counterparts in the OS/2 batch language (the Setlocal and Endlocal statements), the REXX SETLOCAL and ENDLOCAL functions can be nested.

Applications Programming Interface Functions

The following built-in REXX functions can be used in a REXX program to register, drop or query external function packages and to create and manipulate external data queues.

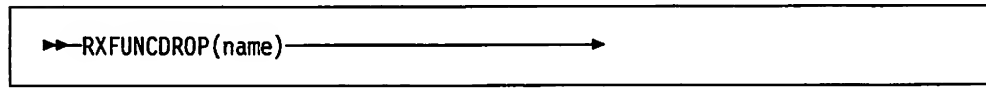
- See "External Functions" on page 167 for a full discussion of the external-function interfaces.
- See "Queue Interface" on page 144 for a full discussion of applications queuing services.

RXFUNCADD



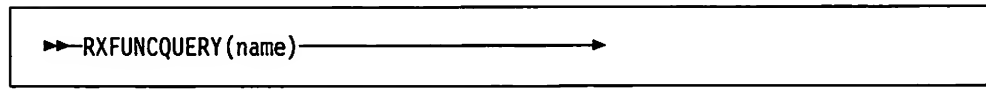
RXFUNCADD registers the function *name*, making it available to REXX procedures. A `0` return value signifies successful registration. See "REXX Interface" on page 175 for more information.

RXFUNCDROP



RXFUNCDROP removes (deregisters) the function *name* from the list of available functions. A `0` zero return value signifies successful removal. See "REXX Interface" on page 175 for more information.

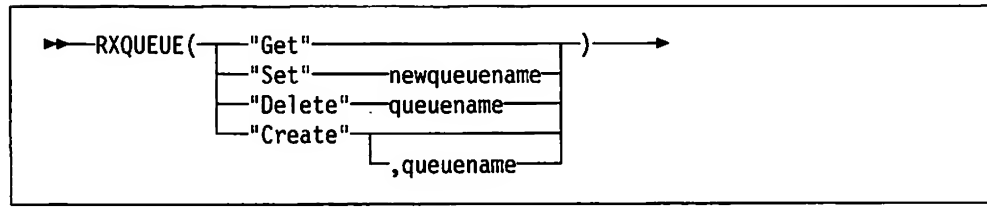
RXFUNCQUERY



RXFUNCQUEUEY queries the list of available functions for a registration of the *name* function. The function returns a value of `0` if the function is registered and a value of 1 if it is not. See "REXX Interface" on page 175 for more information.

Queue Interface

RXQUEUE



RXQUEUE is used in a REXX program to create and delete external data queues and to set and query their names. See "RXQUEUE Function" on page 145 for more information.

Chapter 5. Parsing for PARSE, ARG, and PULL

PARSE, ARG, and PULL allow a selected string to be parsed (split up) and assigned into variables, under the control of a template. The various mechanisms in the template allow a string to be split up into words (delimited by blanks), or by explicit matching of patterns or numeric position—for example to extract data from particular columns of a record read from a file.

This section first gives some informal examples of how to use the parsing template, then describes the mechanisms used.

Introduction

Here are some examples that illustrate how parsing works.

Parsing Words

The simplest form of a parsing template consists of a list of variable names. The data being parsed is split up into words (characters delimited by blanks), and each word from the data is assigned to a variable in sequence. The final variable is treated differently in that it is assigned whatever is left of the original data and may, therefore, contain several words, and possibly leading and trailing blanks.

```
Parse value 'This is a sentence.' with  v1 v2 v3
/* is equivalent to: */
v1 = "This";   v2 = "is";   v3 = "a sentence."
```

In this example, v1 would get the value `This`, v2 would get the value `is`, and v3 would get `a sentence.`

Leading blanks and trailing blanks are removed from each word in the string before the word is assigned to a variable, except for the word or group of words assigned to the last variable. Variables set in this manner (v1 and v2 in the example above) will never have leading or trailing blanks. But the last variable (v3 in the example) could have both leading and trailing blanks, if extra blanks were specified before a or after `sentence.`

For example,

```
Parse value 'This is   a sentence.' with  v1 v2 v3
/* is equivalent to: */
v1 = "This";   v2 = "is";   v3 = "  a sentence."
```

In this example, v1 would get the value `This`, v2 would get the value `is`, and v3 would get `  a sentence.`

In addition, if you use PARSE UPPER (or the ARG or PULL instruction), the whole string is translated into uppercase (that is, lowercase a-z to uppercase A-Z) before parsing begins.

Note that all variables mentioned in a template are always given a new value; if there are fewer words in the data than variables in the template, the unused variables are set to null.

© Copyright IBM Corp. 1983, 1990

Parsing Using String Patterns

You can use a string in a template to split up the data:

```
Parse value 'To be, or not to be?' with  w1 ',' w2
/* causes the data to be scanned for the comma,      */
/* then split at that point, thus:                  */
w1 = "To be";   w2 = " or not to be?"
```

w1 is set to To be, and w2 is set to or not to be?. A string used in this way is called a **pattern**. Note that the pattern itself (and **only** the pattern) is removed from the data. In fact, each section is treated in just the same way as the whole string was in the previous example, and so either section can be split up into words.

```
Parse value 'To be, or not to be?' with  w1 ',' w2 w3 w4
/* is equivalent to: */
w1 = "To be";   w2 = "or";   w3 = "not";   w4 = "to be?"
```

w2 and w3 get the values or and not, and w4 gets the remainder: to be?. If you specified UPPER on the instruction, all the variables would be translated to uppercase.

If the string in these examples did not contain a comma, the pattern would effectively "match" the end of the string: so the variable to the left of the pattern would get the entire input string, and the variables to the right would be set to null. Note that a null string is never found; it always matches the end of the string.

You can specify the pattern as a variable by putting the variable name in parentheses. The following instructions, therefore, have the same effect as the last example:

```
comma=','
Parse value 'To be, or not to be?' with  w1 (comma) w2 w3 w4
```

Parsing Using Numeric Patterns

The third type of parsing mechanism is the numeric pattern. This works in the same way as the string pattern except that it specifies a column number. So:

```
Parse value 'Flying pigs have wings' with  x1 5 x2
/* splits the data at column 5.  Equivalent to */
x1 = "Flyi";   x2 = "ng pigs have wings"
```

splits the data at column 5, and x1 becomes Flyi and x2 starts at column 5 and becomes ng pigs have wings.

More than one pattern is allowed, so for example:

```
Parse value 'Flying pigs have wings' with  x1 5 x2 10 x3
/* splits the data at columns 5 and 10.  Equivalent to */
x1 = "Flyi";   x2 = "ng pi";  x3 = "gs have wings"
```

splits the data at columns 5 and 10, and x2 becomes ng pi and x3 becomes gs have wings.

The numbers can be relative to the last number used, so

```
Parse value 'Flying pigs have wings' with  x1 5 x2 +5 x3
```

has exactly the same effect as the last example: here the +5 can be thought of as specifying the length of the data to be assigned to x2.

As with literal string patterns, you can specify a positional pattern as a variable by putting the name of a variable in parentheses, in place of a number. You can

indicate an absolute column number with an equal sign (=) instead of using a plus or minus sign. The last example would, therefore, be:

```
start=5
length=5
data='Flying pigs have wings'
parse var data  x1 =(start) x2 +(length) x3
```

String patterns and numeric patterns can be mixed (in effect the beginning of a string pattern just specifies a variable column number) and some very powerful things can be done with templates. The "Definition" section (following) describes in more detail how the various mechanisms interact.

Parsing Arguments

Finally, it is possible to parse more than one string. For example, an internal function can have more than one argument string. To get at each string in turn, you just put a comma in the parsing template. For example, if the invocation of the function "FRED" was:

```
fred('This is the first string',2)
```

the instruction

```
PARSE ARG first, second
/* is equivalent to */
first = "This is the first string";   second = "2"
```

The variable `first` contains the string "This is the first string". The variable `second` contains the string "2". Between the commas you can put a normal template, with patterns, and so forth, to do more complex parsing on each of the argument strings.

Definition

This section describes the rules that govern parsing.

In its most general form, a template consists of alternating pattern specifications and variable names. The pattern specifications and variable names are used strictly in sequence from left to right, and are used once only. In practice, various simpler forms are used in which either variable names or patterns can be omitted; we can, therefore, have variable names without patterns in between, and patterns without intervening variable names.

In general, the value assigned to a variable is that sequence of characters in the input string between the point that is matched by the pattern on its left and the point that is matched by the pattern on its right.

If the first item in a template is a variable, there is an implicit pattern on the left that matches the start of the string, and similarly if the last item in a template is a variable, there is an implicit pattern on the right that matches the end of the string. Therefore, the simplest template consists of a single variable name, which, in this case, is assigned the entire input string.

Setting a variable during parsing is identical with setting a variable in an assignment. It is, therefore, possible to set an entire collection of compound variables during parsing. When a variable follows another variable, the action taken is the same for all kinds of patterns; this action is described under "Parsing Strings into Words" on page 118.

The constructs that appear as patterns fall into two categories:

- String patterns that act by searching for a matching string
- Numeric (positional) patterns that specify an absolute or relative position in the data.

Both string and numeric patterns can be variable patterns, which means that you can specify a pattern by using the value of a variable instead of a literal string or number.

For the following examples, assume that the following string is being parsed (note that all blanks are significant):

```
'This is  the data which, I think,  is scanned.'
```

Parsing Strings into Words

If a variable is followed by another variable, a special action is taken. This is similar to the pattern ' ' (a single blank) being between them, except that leading blanks at the current position in the input data are skipped over before the search for the next blank takes place. This means that the value assigned to the left-hand variable is the next word in the string and has neither leading nor trailing blanks.

Thus, the template:

```
w1 w2 w3 rest ','
```

results in:

```
w1   = "This"
w2   = "is"
w3   = "the"
rest = "data which"
```

Note that the final variable (rest in this example) could have had both leading blanks and trailing blanks, because only the blank that delimits the previous word is removed from the data.

Also observe that this example is not the same as specifying explicit blanks as patterns, as the template:

```
w1 ' ' w2 ' ' w3 ' ' rest ','
```

(in fact) results in:

```
w1   = "This"
w2   = "is"
w3   = "" (null)
rest = "the data which"
```

because the third pattern would match the third blank in the data.

Note: Quotation marks are not part of the value. They are shown here and in following examples only to indicate leading or trailing blanks.

In general then, when a variable is followed by another variable, parsing of the input by tokenization into words is implied.

Parsing with Literal String Patterns

Literal patterns cause scanning of the input data string to find a sequence that matches the value of the literal. Literals are expressed as a quoted string.

When the template:

```
w1 ',' w2 ',' rest
```

is used to parse the example string, the result is:

```
w1   = "This is  the data which"
w2   = " I think"
rest = "  is scanned."
```

Here the string is parsed using a template that asks that each of the variables receive a value corresponding to a portion of the original string between commas; the commas are given as quoted strings. Note that the patterns (in this example, the commas) themselves are removed from the data being parsed.

A different parse would result with the template:

```
w1 ',' w2 ',' w3 ',' rest
```

which would result in:

```
w1   = "This is  the data which"
w2   = " I think"
w3   = "  is scanned."
rest = "" (null)
```

This illustrates an important rule. When a match for a pattern cannot be found in the input string, it instead "matches" the end of the string. Thus, no match was found for the third ',' in the template, and so `w3` was assigned the rest of the string. Because the pattern on its left had already reached the end of the string, `rest` was assigned a null value.

A null pattern (a string of length 0) can be used to match the end of the data explicitly. This is mainly useful with positional patterns (described later).

Note that *all* variables that appear in a template are assigned a new value.

Use of the Period as a Placeholder

The symbol consisting of a single period acts as a placeholder in a template. It has exactly the same effect as a variable name, except that no variable is set. It is especially useful as a "dummy variable" in a list of variables or to collect unwanted information at the end of a string. Thus, when the template:

```
. . . word4 .
```

is used to parse the same example string:

```
'This is  the data which, I think,  is scanned.'
```

the result is:

```
word4 = "data"
```

That is, the fourth word (data) is extracted from the string and placed in the variable word4.

Parsing with Positional (Numeric) Patterns

Positional patterns can be used to cause the parsing to occur on the basis of position within the string, rather than on its contents. They take the form of whole numbers. A plus, minus, or equal sign before a number indicates relative or absolute positioning; this is optional. A matching operation can "back up" to an earlier position in the data string only when you use positional patterns.

Absolute positional patterns: When no sign or an equal sign precedes a number in a template, this is an absolute positional pattern. The number refers to a particular (absolute) character column in the input, with 1 referring to the first column. For example, when the template

```
s1 10 s2 20 s3
```

is used to parse the example string, this results in

```
s1 = "This is  "
s2 = "the data w"
s3 = "hich, I think,  is scanned."
```

Here s1 is assigned characters from the first through the ninth character, and s2 receives input characters 10 through 19. The final variable, s3, is assigned the remainder of the input. You can place an equal sign before the number to indicate that it is an absolute column position. This has the same result as the preceding example:

```
s1 =10 s2 =20 s3
```

Relative positional patterns: When a plus or minus sign precedes a number in a template, this is a relative positional pattern. The number indicates movement relative to the character position at which the previous pattern match occurred.

If you specify a number with a plus or minus sign, the position used for the next match is calculated by adding or subtracting the number given to the last matched position. The **last matched position** is the position of the first character of the last match, whether specified numerically or by a string. For example, the instructions:

```
a = '123456789'
parse var a  3 w1 +3 w2 3 w3
```

result in:

```
w1 = "345"
w2 = "6789"
w3 = "3456789"
```

The +3 in this case is equivalent to the absolute number 6 in the same position and specifies the length of the data to be assigned to the variable w1.

This example also illustrates the effects of a pattern that implies movement to a character position to the left of, or to the point where matching has already occurred. Movement is from column 6, the starting position for w2, to column 3, the starting position for w3. The variable on the left is assigned characters through the end of the input, and the variable on the right is, as usual, assigned characters starting at the position dictated by the pattern.

The following PARSE instruction assigns the same values to w1, w2, and w3 as above:

```
a = '123456789'
parse var a  3 w1 +3 w2 -3 w3
```

3 specifies the starting position for w1, column 3. +3 tells you to move 3 positions to the right of the starting position of w1. This is the starting position of w2, column 6. -3 tells you to move 3 positions to the left of the starting position of w2. This is the starting position of w3, column 3.

This is useful for making multiple assignments:

```
parse var x 1 w1 1 w2 1 w3
```

assigns the (entire) value of x to w1, w2, and w3. (The first "1" here could be omitted as it is effectively the same as the implicit starting pattern described at the beginning of this section.)

If a positional pattern specifies a column that is greater than the length of the data, it is equivalent to specifying the end of the data (that is, no padding takes place). Similarly, if a pattern specifies a column to the left of the first column of the data, this is not an error but instead is taken to specify the first column of the data.

Any pattern match sets the "last position" in a string to which a relative positional pattern can refer. The "last position" set by a literal pattern is the position at which the match occurred; that is, the position in the data of the *first* character in the pattern. The *first* character in this case is not removed from the parsed data. Thus, the template:

```
',' -1 x +1
```

1. Finds the first comma in the input (or the end of the string if there is no comma).
2. Backs up one position.

3. Assigns one character (the character immediately preceding the comma or end of string) to the variable x.

A possible application of this is looking for abbreviations in a string. Thus, the instruction:

```
/* Ensure options have leading blank and are uppercase */
parse upper value ' 'opts with ' PR' +1 prword ' '
```

sets the variable prword to the first word in opts that starts with PR or sets it to null if no such word exists. Note that +0 is a valid positional pattern.

When a literal pattern is followed by a signed (+/-) positional pattern, the literal string IS NOT REMOVED from the data being parsed. Instead it is parsed into the first variable following the literal pattern. Thus, the following two cases:

```
a='This is the data which, I think, is scanned.'

   CASE 1:  parse var a 'which' +5 y
   CASE 2:  parse var a 'which' x +5 y
```

result in:

```
   CASE 1:  y = ", I think, is scanned."
   CASE 2:  x = "which"
            y = ", I think, is scanned."
```

Note: If a number in a template is preceded by a "+" or a "-," this is taken to be a signed positional pattern. There can be blanks between the sign and the number, since initial scanning removes blanks adjacent to special characters.

Parsing with Variable Patterns

It is sometimes desirable to be able to specify a pattern by using the value of a variable instead of a fixed string or number. You can do this by placing the name of the variable to serve as the pattern in parentheses. This is called a **variable reference.** Blanks are not necessary inside or outside the parentheses, but you can add them if you wish.

If an equal, plus, or minus sign does not precede the parenthesis to the left of the variable name, the value of the variable is then treated as a literal (string) pattern. The variable can be one that has been set earlier in the parsing process, so, for example:

```
input="L/look for/1 10"
parse var input  verb 2 delim +1 string (delim) rest
```

sets:

```
   verb   = "L"
   delim  = "/"
   string = "look for"
   rest   = "1 10"
```

If an equal, plus, or minus sign precedes the left parenthesis, then the value of the variable is treated as an absolute or relative positional pattern, instead of as a literal string pattern. In this case, the value of the variable must be a nonnegative whole number. It can also be one set up earlier in the parsing process.

In the following example, a file has variable length data fields containing a name and address; each record starts with two numbers specifying the starting columns of the name and address.

```
dataline = '10 30    Mary Ellen Friedmann123Main Street'
PARSE VAR dataline,
   namecol addrcol =(namecol) name =(addrcol) address
```

In the following example, each record starts with two numbers specifying the starting column of the name and the width of the name, and the address immediately follows the name.

```
PARSE VAR dataline,
   namecol namewidth =(namecol) name +(namewidth) address
```

Parsing Multiple Strings

A parsing template can parse **multiple strings** if you use the special pattern comma (,) in the template. Each comma is an instruction to the parser to move on to the next string. Other patterns and variables can be specified for each string parsed, as usual. The only time multiple strings are available is in the ARG (or PARSE ARG) instruction. When an internal function or subroutine is invoked it can have several argument strings, and a comma is used to access each in turn. Thus, the template:

```
word1 string1, string2, num
```

puts the first word of the first argument string into word1, the rest of that string into string1, and the next two strings into string2 and num. If insufficient strings are specified in the invocation, unused variables are set to null. Similarly, if only one string is available (as on the other PARSE variations), then any variables that follow a comma pattern are set to null.

Chapter 6. Numerics and Arithmetic

REXX defines the usual arithmetic operations (addition, subtraction, multiplication, and division) in as *natural* a way as possible. This means that the rules followed are those that are conventionally taught in schools and colleges.

During the design of these facilities, however, it was found that unfortunately the rules used vary considerably (indeed much more than generally appreciated) from person to person and from application to application and in ways that are not always predictable. The arithmetic described here is therefore a compromise that (although not the simplest) should provide acceptable results in most applications.

Introduction

Numbers (that is, character strings used as input to REXX arithmetic operations) can be expressed very flexibly. Leading and trailing blanks are permitted, and exponential notation can be used. Some valid numbers are:

```
   12          /* an integer                */
  -76          /* signed integer            */
   12.76       /* decimal places            */
' +  0.003 '   /* blanks around the sign etc */
   17.         /* same as "17"              */
     .5        /* same as "0.5"             */
   4E9         /* exponential notation      */
    0.73e-7    /* exponential notation      */
```

(Exponential notation means that the number includes a power of ten following an E that indicates how the decimal point should be shifted. Thus `4E9` is a short way of writing 4000000000, and `0.73e-7` is short for 0.000000073.)

Arithmetic operators include addition (+), subtraction (−), multiplication (*), power (**), division (/), prefix plus (+), and prefix minus (−). In addition, there are two further division operators, integer divide (%) divides and returns the integer part, remainder (//) divides and returns the remainder.

The result of an arithmetic operation is formatted as a character string according to definite rules. The most important of these rules are as follows (see "Definition" on page 126 for further details).

- Results are calculated up to some maximum number of significant digits (the default is 9, but you can alter this with the NUMERIC DIGITS instruction to give whatever accuracy you need). Thus if a result requires more than 9 digits, it would normally be rounded to 9 digits. For example, the division of 2 by 3 would result in 0.666666667 (it would require an infinite number of digits for perfect accuracy).
- Except for division and power, trailing zeros are preserved (this is in contrast to most popular calculators, which remove all trailing zeros). For example:

  ```
  2.40 + 2    ->    4.40
  2.40 - 2    ->    0.40
  2.40 * 2    ->    4.80
  2.40 / 2    ->    1.2
  ```

  This behavior is desirable for most calculations (especially financial calculations).

© Copyright IBM Corp. 1983, 1990

If necessary, you can remove trailing zeros with the STRIP function, see "STRIP" on page 100, or by division by 1.

- A zero result is always expressed as the single digit 0.
- Exponential form is used for a result depending on the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number will be expressed in exponential notation:

```
1e6 * 1e6     ->    1E+12
            /* not 1000000000000 */
  1 / 3E10    ->    3.33333333E-11
            /* not 0.000000000333333333 */
```

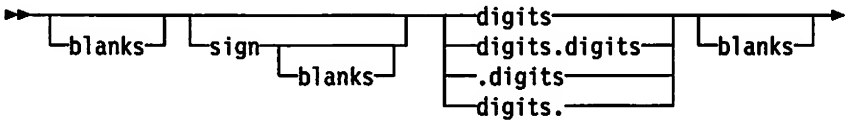
Definition

A precise definition of the arithmetic facilities of the REXX language is given here.

Numbers

A *number* in REXX is a character string that includes one or more decimal digits, with an optional decimal point. The decimal point can be embedded in the number, or can be prefixed or suffixed to it. The group of digits (and optional decimal point) constructed this way can have leading or trailing blanks and an optional sign (+ or −) that must come before any digits or decimal point. The sign can also have leading or trailing blanks.

Therefore, *number* is defined as:



Where:

sign Is either '+' or '−'

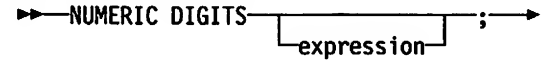
blanks Are one or more spaces

digits Are one or more of the decimal digits 0 through 9.

Note that a single period alone is not a valid number.

Precision

The maximum number of significant digits that can result from an operation is controlled by the instruction:



expression is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations are carried out. Results are rounded to that precision, if necessary.

If you do not specify expression in this instruction, or if a NUMERIC DIGITS instruction has not been executed since the start of a program, the default precision is used. The REXX standard for the default precision is 9.

Note that NUMERIC DIGITS can set values below the default of nine. Use small values, however, with care—the loss of precision and rounding requested thus affects all REXX computations, including, for example, the computation of new values for the control variable in DO loops.

Arithmetic Operators

REXX arithmetic is affected by the operators + , − , * , / , % , // , and ** , which all act on two terms, and the prefix plus and minus operators, which both act on a single term. This section describes the way in which these operations are carried out.

Before every arithmetic operation, the term or terms being operated upon have leading zeros removed (noting the position of any decimal point, and leaving just one zero if all the digits in the number are zeros). They are then truncated (if necessary) to DIGITS + 1 significant digits (the extra digit is a *guard digit*) before being used in the computation. The operation is then carried out to less than twice that precision, as described under the individual operations that follow. When the operation is completed, the result is rounded if necessary to the precision specified by the NUMERIC DIGITS instruction.

Every operation is carried out in such a way that no errors will be introduced except during the final rounding of the result to the specified significance. (That is, input data is first truncated to the appropriate significance (NUMERIC DIGITS+1) before being used in the computation, and then divisions and multiplications are carried out to double that precision, as needed.)

Rounding is done in the *traditional* manner. The digit to the right of the least significant digit in the result (the *guard digit*) is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Even or odd rounding would require the ability to calculate to arbitrary precision at all times and is therefore not the mechanism defined for REXX.

A conventional zero is supplied in front of the decimal point if otherwise there would be no digit before it. Significant trailing zeros are retained for addition, subtraction, and multiplication, according to the rules that follow, except that a result of zero is always expressed as the single digit 0. For division, trailing zeros are removed after rounding.

The "FORMAT," see page 86 , allows a number to be represented in a particular format if the standard result provided does not meet your requirements.

Arithmetic Operation Rules—Basic Operators

The basic operators (addition, subtraction, multiplication, and division) operate on numbers as follows. All numbers have insignificant leading zeros removed before being used in computation.

Addition and Subtraction

If either number is zero, the other number, rounded to NUMERIC DIGITS digits if necessary, is used as the result (with sign adjustment as appropriate). Otherwise, the two numbers are extended on the right and left as necessary up to a total maximum of DIGITS + 1 digits (the number with the smaller absolute value may therefore lose some or all of its digits on the right) and are then added or subtracted as appropriate.

Example:

```
xxx.xxx  +  yy.yyyyy
```

becomes:

```
   xxx.xxx00
+  0yy.yyyyy
-------------
   zzz.zzzzz
```

The result is then rounded to the current setting of NUMERIC DIGITS if necessary, taking into account any extra *carry* digit on the left after addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted, and any insignificant leading zeros are removed.

The *prefix operators* are evaluated using the same rules; the operations *+number* and *−number* are calculated as *0+number* and *0−number*, respectively.

Multiplication

The numbers are multiplied together (*long multiplication*) resulting in a number that can be as long as the sum of the lengths of the two operands.

Example:

```
xxx.xxx * yy.yyyyy
```

becomes:

```
zzzzz.zzzzzzzz
```

The result is then rounded, counting from the first significant digit of the result, to the current setting of NUMERIC DIGITS.

Division

For the division:

```
yyy / xxxxx
```

the number yyy is extended with zeros on the right until it is larger than the number xxxxx (with note being taken of the change in the power of ten that this implies). Thus, in this example, yyy becomes yyy00. Long division then takes place. This might be written:

```
        zzzz
     ┌──────
xxxxx │ yyy00
```

The length of the result zzzz is such that the rightmost z is at least as far right as the rightmost digit of the *extended* y number in the example. During the division, the y number is extended further as necessary. The z number may increase up to

NUMERIC DIGITS+1 digits, at which point the division stops and the result is rounded. Following completion of the division (and rounding if necessary), insignificant trailing zeros are removed.

Basic Operator Examples

The following are some examples that illustrate the main implications of the rules described previously:

```
/* With:  Numeric digits 5 */
12+7.00     ->    19.00
1.3-1.07    ->     0.23
1.3-2.07    ->    -0.77
1.20*3      ->     3.60
7*3         ->    21
0.9*0.8     ->     0.72
1/3         ->     0.33333
2/3         ->     0.66667
5/2         ->     2.5
1/10        ->     0.1
12/12       ->     1
8.0/2       ->     4
```

Notes:

1. With all the basic operators, the position of the decimal point in the terms being operated upon is arbitrary. The operations may be carried out as integer operations with the exponent being calculated and applied afterwards. Therefore the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.
2. In the previous examples, the position of the decimal point is arbitrary. In fact the operations may be carried out as integer operations with the exponent being calculated and applied after the operations. Therefore none of the operations is in any way dependent on the position of the decimal point, and hence results are completely independent of the number of decimal places.

Arithmetic Operators—Additional Operators

The power (**), integer divide (%), and remainder (//) operators rules follow.

Power

The ** operator raises a number to a whole power, which can be positive or negative. If negative, the absolute value of the power is used, and then the result is inverted (divided into 1). For calculating the result, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by one).

In practice (see note 1 on page 130 for rationale), the result is calculated by the process of left-to-right binary reduction. For x**n, n is converted to binary, and a temporary accumulator is set to 1. If n = 0 the calculation is complete. (Thus, x**0 = 1 for all x, including 0**0.) Otherwise each bit (starting at the first nonzero bit) is inspected from left to right. If the current bit is 1, the accumulator is multiplied by x. If all bits have now been inspected, the calculation is complete; otherwise the accumulator is squared and the next bit is inspected for multiplication. When the calculation is complete, the temporary result is ready for division by or into 1 to provide the final answer. The multiplication and division are done under the normal REXX arithmetic combination rules with the initial calculation (the multiplications) using precision of DIGITS + L + 1 digits (where L is the length in digits of the whole number n) and the final division using the usual NUMERIC

DIGITS digits. The precision specified for the intermediate calculations ensures that the final result will differ by at most 1, in the least significant position, from the *true* result. Half of this maximum error comes from the intermediate calculation, and half from the final rounding.

Integer Division

The % operator divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used, the sign of the final result is the same as that which would result if normal division were used.

The result returned will have no fractional part (that is, no decimal point or zeros following it). If the result cannot be expressed by digits within the precision set by the NUMERIC DIGITS instruction, the operation is in error and will fail. For example, 10000000000%3 requires 10 digits for the result (3333333333) and would therefore fail if NUMERIC DIGITS 9 were in effect.

Remainder

The // operator returns the remainder from integer division, which is defined as being the residue of the dividend after the operation of calculating integer division as previously described. The sign of the remainder, if nonzero, is the same as that of the original dividend.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated).

Additional Operator Examples

Following are some examples using the power, integer divide, and remainder operators previously described:

```
/* Again with:  Numeric digits 5 */
2**3        ->     8
2**-3       ->     0.125
1.7**8      ->    69.758
2%3         ->     0
2.1//3      ->     2.1
10%3        ->     3
10//3       ->     1
-10//3      ->    -1
10.2//1     ->     0.2
10//0.3     ->     0.1
```

Notes:

1. A particular algorithm for calculating powers is used, since it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It therefore gives better performance than the simpler definition of repeated multiplication. Since results may differ from those of repeated multiplication, the algorithm is defined here.
2. The integer divide and remainder operators are defined so that they can be calculated as a by-product of the standard division operation. The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.

Numeric Comparisons

The comparison operators are listed on page 14. You can use any of these for comparing numeric strings. However, you should not use = =, \= =, ¬ = =, > >, \> >, ¬ > >, < <, \< <, and ¬ < < to compare numeric values because leading or trailing blanks and leading zeros are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. For example, the operation:

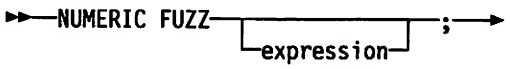
```
A ? B
```

where ? is any numeric comparison operator, is identical to:

```
(A - B) ? '0'
```

It is therefore the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

Comparison of two numbers is affected by a quantity called *FUZZ*, which is set by the instruction:



Here expression must result in a whole number that is zero or positive. This FUZZ number controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each numeric comparison operation. That is, the numbers are subtracted under a precision of DIGITS – FUZZ digits during the comparison. Clearly FUZZ must be less than DIGITS.

Thus if DIGITS = 9, and FUZZ = 1, the comparison is carried out to 8 significant digits, just as though NUMERIC DIGITS 8 had been put in effect for the duration of the operation.

Examples:

```
Numeric digits 5
Numeric fuzz 0
say  4.9999 = 5     /* would display 0    */
say  4.9999 < 5     /* would display 1    */

Numeric fuzz 1
say  4.9999 = 5     /* would display 1    */
say  4.9999 < 5     /* would display 0    */
```

Exponential Notation

The previous numbers describe *pure* numbers, in the sense that the character strings that describe numbers could be very long. For example:

```
10000000000 * 10000000000
                would give 100000000000000000000
```

and

```
.00000000001 * .00000000001
                would give 0.000000000000000000001
```

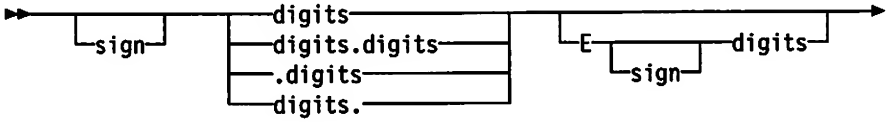
For both large and small numbers some form of exponential notation is useful, both to make long numbers more readable, and to make execution possible in extreme cases. In addition, exponential notation is used whenever the *simple* form would give misleading information.

For example:

```
numeric digits 5
say 54321*54321
```

would display 2950800000 if long form were used. This is clearly misleading, and so the result is expressed as 2.9508E+9 instead.

The definition of *numbers* is therefore extended as (note that blanks are shown only for readability):



The integer following the E represents a power of ten that is to be applied to the number; and the E can be uppercase or lowercase.

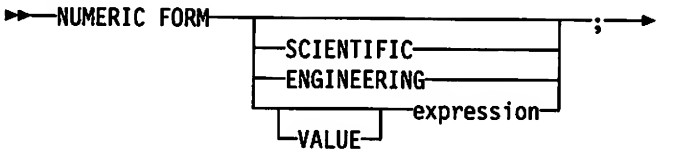
The following are some examples:

```
12E11  =  1200000000000
12E-5  =  0.00012
-12e4  =  -120000
```

The previous numbers are valid for input data at all times. The results of calculations are returned in either conventional or exponential form depending on the setting of DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, exponential form is used. The exponential form generated by REXX always has a sign following the E in order to improve readability. An exponential part of E+0 will never be generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in *long* form by using the FORMAT built-in function, described on page 86.

You can control whether Scientific or Engineering notation is to be used by using the instruction:



The default setting of FORM is SCIENTIFIC.

Scientific notation adjusts the power of ten so there is a single nonzero digit to the left of the decimal point. Engineering notation causes powers of ten to always be expressed as a multiple of 3, the integer part may therefore range from 1 through 999.

```
/* after the instruction */
Numeric form scientific

123.45 * 1e11     ->     1.2345E+13

/* after the instruction */
Numeric form engineering

123.45  * 1e11    ->    12.345E+12
```

Whole Numbers

Within the set of numbers REXX understands, it is useful to distinguish the subset defined as *whole numbers.* A *whole number* in REXX is a number that has a decimal part that is all zeros (or that has no decimal part). In addition, it must be possible to express its integer part simply as digits within the precision set by the NUMERIC DIGITS instruction. REXX would express larger numbers in exponential notation, after rounding, and, hence, these could no longer be safely described or used as *whole numbers.*

Numbers Used Directly by REXX

As discussed, numbers are always rounded (if necessary) according to the setting of NUMERIC DIGITS during any arithmetic operation. Similarly, when REXX directly uses a number, which has not necessarily been involved in an arithmetic operation, the same rounding is also applied.

In the following cases, the number used must be a whole number with the following limits:

Power values (right hand operand of the power operator)	999999999
Values of exprr and exprf in the DO instruction	The current numeric precision (up to 999999999).
Values given for DIGITS or FUZZ in the NUMERIC instruction	999999999 (Note: FUZZ must always be less than DIGITS).
Positional patterns in parsing templates	4 billion (+4.0E9)
Number given for option in the TRACE instruction.	4 billion (+4.0E9)

Errors

Two types of errors may occur during arithmetic:

- Overflow or Underflow

  This error occurs if the exponential part of a result would exceed the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. The maximum for exponents in the OS/2 program implementation is 999999999.

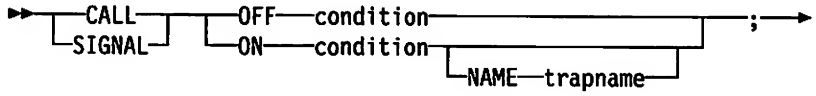
  Since this allows for large exponents, overflow or underflow is treated as a terminating *syntax* error.

- Insufficient storage

  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered a terminating error as usual, rather than an arithmetical error.

Chapter 7. Conditions and Condition Traps

CALL and SIGNAL modify the flow of execution in a REXX program by using condition traps. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions (see "CALL" on page 28 and "SIGNAL" on page 58).



The single symbols, `condition` and `trapname`, are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

If a condition trap is enabled and the specified condition occurs, control passes to the routine or label *trapname*. SIGNAL or CALL is used, depending on whether the most recent trap for the condition was set using SIGNAL ON or CALL ON respectively.

The conditions and their corresponding events, which can be trapped, are:

ERROR
Raised if any host command indicates an error condition upon return. It is also raised if any command indicates failure and CALL ON FAILURE or SIGNAL ON FAILURE is not set. The condition is raised at the end of the clause that invoked the command, but is ignored if the ERROR condition trap is already in the delayed state.

Note: In VM and TSO/E, SIGNAL ON ERROR traps all positive return codes, and negative return codes only if CALL ON FAILURE and SIGNAL ON FAILURE are not set.

FAILURE
Raised if any host command indicates a failure condition upon return. The condition is raised at the end of the clause that invoked the command, but is ignored if the FAILURE condition trap is already in the delayed state.

For the default CMD processor, an attempt to issue an unknown command will raise a FAILURE condition. An attempt to issue a command to an unknown subcommand environment will also raise a FAILURE condition.

Note: In VM and TSO/E, SIGNAL ON FAILURE traps all negative return codes from commands.

HALT
Raised if an external attempt is made to interrupt execution of the program.

This option is not implemented for CMD program files running on the OS/2 program; in such files, the command SIGNAL ON HALT is ignored.

Note: Application programs that use the REXX language processor may use the RXHALT exit to halt execution of a REXX macro. See "System Exits" on page 191.

© Copyright IBM Corp. 1983, 1990

NOTREADY
Raised if an error occurs during an input or output operation. See "Errors During Input and Output" on page 147.

NOVALUE
Raised if an uninitialized variable is used:

- As a term in an expression
- As the name following the VAR subkeyword of the PARSE instruction
- As an unassigned variable pattern in a parsing template.

This condition may only be specified for SIGNAL ON.

SYNTAX
Raised if an interpretation error is detected. This condition may only be specified for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON or OFF, and any trap name) of that condition trap. Thus, a SIGNAL ON HALT replaces any current CALL ON HALT, and so on.

Action Taken When a Condition Is Trapped

When a condition trap is currently enabled (ON has been specified), the trap is in effect. So, when the specified condition occurs, instead of the usual flow of control a CALL *trapname* or SIGNAL *trapname* option is executed automatically (that is, passes control to a label or routine). The label or routine given control depends on whether you used the NAME *trapname* option when you enabled the condition trap.

If you did not explicitly specify a *trapname*, control is passed to the label or routine that matches the name of the *condition* itself (ERROR, FAILURE, HALT, NOVALUE, or SYNTAX).

If you specified *trapname* after the NAME subkeyword of the CALL ON or SIGNAL ON instruction, control is passed to the label or routine specified, rather than the name of the *condition*.

The sequence of events, once a condition has been trapped, varies depending on whether SIGNAL or CALL is executed:

- If the action taken is SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and the SIGNAL takes place in exactly the same way as usual, see page 58.

  If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it once the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, then if the SIGNAL ON SYNTAX label name is not found a normal syntax error termination occurs.

- If the action taken is CALL, CALL is made in the usual way, see page 28, except that the special variable RESULT is not affected by the call. If the routine should RETURN any data, then the returned character string is ignored.

  Note that CALL ON can only occur at clause boundaries. Because these conditions (ERROR, FAILURE, and HALT) can arise during execution of an INTERPRET instruction, execution of the INTERPRET may be interrupted and later resumed if CALL ON was used.

  Before the CALL is made, the condition trap is put into a *delayed* state. This state persists until the RETURN from the CALL, or until an explicit CALL (or

SIGNAL) ON (or OFF) is made for the condition. This delayed state prevents a premature condition trap at the start of the routine called to process a condition trap. When a condition trap is in the delayed state it remains enabled, but if the condition is trapped again any action (including the updating of the condition information) is delayed until one of the following events:

- CALL ON or SIGNAL ON, for the delayed condition, is executed. In this case CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been executed.
- CALL OFF or SIGNAL OFF, for the delayed condition, is executed. In this case the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.
- RETURN is made from the subroutine. In this case the condition trap is no longer delayed and the subroutine is called again immediately.

On RETURN from the CALL, the original flow of execution is resumed (that is, the flow is not affected by the CALL).

Notes:

1. In all cases, the condition is raised (and the current instruction terminated) immediately upon detection of the error. Therefore, the instruction during which an event occurs may be only partly executed. For example, if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment does not take place. Note that ERROR, FAILURE, and HALT can only occur at clause boundaries, but could arise in the middle of an INTERPRET instruction.
2. The state (ON, OFF, or DELAY, and any *trapname*) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See "CALL" on page 28 for details of other information that is saved during a subroutine call.
3. The state of condition traps is not affected when an external routine is invoked by a CALL, even if the external routine is a REXX program. On entry to any REXX program, all condition traps have an initial setting of OFF.
4. While user input is executed during interactive tracing, all conditions are set OFF so that unexpected transfer of control does not occur should, for example, the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive tracing does not cause exit from the program, but is trapped specially and then ignored after a message is given.
5. Certain execution errors are detected by the host interface either before execution of the program starts or after the program has exited. SIGNAL ON SYNTAX. cannot trap these errors.

Note that *labels* are clauses consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, multiple labels are allowed before another type of clause.

Condition Information

When any condition is trapped and causes SIGNAL (or CALL), this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the CONDITION built-in function (see "CONDITION" on page 79).

The condition information includes:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction executed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition.

The descriptive string varies, depending on the condition trapped. In the case of SIGNAL, the descriptive string that is passed to the external environment as a result of the command is one of the following:

ERROR The string that was processed and resulted in the error condition.

FAILURE The string that was processed and resulted in the failure condition.

HALT Any string associated with the halt request. This can be the null string if no string was provided.

NOTREADY The name of the stream being manipulated when the error occurred and the NOTREADY condition was raised.

NOVALUE The derived name of the variable whose attempted reference caused the NOVALUE condition.

SYNTAX Any string the language processor associated with the error. This can be the null string if no specific string is provided. Note that the special variable RC and SIGL provide information on the nature and position of the processing error.

The current condition information is replaced when control is passed to a label as the result of a condition trap (CALL ON or SIGNAL ON). Condition information is saved and restored across subroutine or function calls, including one due to a CALL ON trap. A routine invoked by CALL ON, therefore, can access the appropriate condition information. Any previous condition information is still available after the routine returns.

The Special Variable SIGL

When any transfer of control due to a SIGNAL (or CALL) takes place, the line number of the clause currently running is stored in the REXX special variable SIGL. This is especially useful for SIGNAL ON SYNTAX when the number of the line in error can be used, for example, to control an editor. Typically, code following the SYNTAX label may PARSE SOURCE to find the source of the data, then invoke an editor to edit the source file positioned at the line in error. Note that in this case the program has to be reinvoked before any changes made in the editor can take effect.

Alternatively, SIGL can be used to help determine the cause of an error, such as the occasional failure of a function call as in the following example:

```
/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  errormsg='REXX error' rc 'in line' sigl':' errortext(rc)
  say errormsg
  say sourceline(sigl)
  trace '?r'; nop
```

This code first displays the error code, line number, and message text. It then displays the line in error, and finally drops into debug mode to let you to inspect the values of the variables used at the line in error.

The Special Variable RC

For ERROR and FAILURE, the REXX special variable RC is set to the command return code error number before control is transferred to the condition label. For SIGNAL ON SYNTAX, RC is set to the syntax error number.

Chapter 8. Input and Output Streams

REXX defines only simple, character oriented, forms of input and output. In general, communication to or from the user is in the form of a stream of characters. These streams may be manipulated either character-by-character or line-by-line. In addition to these character streams, an external data queue is defined for inter-program communication. This queue can only be accessed on a line-by-line basis.

In this discussion, input and output will be described as though communicating with a human user, but a character stream might, in fact, have a variety of sources or destinations. These may include files, serial interfaces, displays, or networks. A character stream may therefore be:

- **transient**, or dynamic; for example, data sent or received over a serial interface

  or

- **persistent**, in a static form; for example, a file or data object.

The form that stream names can have is defined by the OS/2 program. For example, a stream can be a file, in which case the stream name is any valid drive, path, and file specification for that file. The OS/2 environment provides REXX with one default input stream and one default output stream. These are referred to as STDIN: (the standard input stream) and STDOUT: (the standard output stream). Where no name is provided for a function, the appropriate default stream is assumed. A stream can also be the name of an OS/2 device, including:

STDERR:	Standard error output
CON:	Display screen input and output
KBD:	Keyboard input
PRN:	Current default printer output
LPT1:/LPT2:	Printer devices
COM1:/COM2:	Communication ports
QUEUE:	REXX external data queue.

Notes:

1. For additional information on specific devices, refer to the MODE command in the *OS/2 Commands Reference*.
2. While the form of a stream name is necessarily defined by the OS/2 environment, it is still possible to write programs that use input and output functions and yet are effectively independent of the OS/2 program.

The Input and Output Model

The model of input and output for REXX consists of three logically distinct parts, namely:

- One or more character input streams
- One or more character output streams
- One or more external data queues.

© Copyright IBM Corp. 1983, 1990

These three elements are manipulated by the REXX instructions and built-in routines as follows.

Character Input Streams

Input to REXX programs takes the form of a serial character stream generated by user interaction, or having the characteristics of a stream so generated. Characters may be added to the end of some streams asynchronously; other streams may be static or synchronous.

The instructions that govern the use of input streams are:

- Any named input stream can be read directly as characters by the CHARIN function or as lines by the LINEIN function.
- The default input character stream (STDIN:) can be read as lines by the PULL and PARSE PULL instructions if the external data queue is empty (PULL is the same as PARSE PULL except that uppercase translation takes place).
- The PARSE LINEIN instruction can be used to read lines from the default input character stream regardless of the state of the external data queue. Normally, however, the default input stream is read by using PULL or PARSE PULL.

In a persistent stream, the REXX language processor maintains a current *read position.*

- The CHARS function returns the number of characters currently available in an input character stream from the read position through the end of the stream (including any line-end characters, if these are defined for the stream).
- The LINES function is used to determine if any data remains between the current read position and the end of the stream.
- The read position itself can be manipulated to an arbitrary point in the stream by means of the SEEK command of the STREAM function.

In a transient stream, no read position is available.

- The CHARS and LINES function can determine only if data is present in the stream. For OS/2 devices, the return value is 1 for either of these functions.
- The SEEK command of the STREAM function is not applicable to transient streams.

Character Output Streams

Character output streams provide for output from a REXX program.

- Any output stream can be written in character form with the CHAROUT function.
- Any output stream can be written as lines using the LINEOUT function.
- The default output stream (STDOUT:) can also be written as lines with the SAY instruction.

Both LINEOUT and SAY provide the appropriate line-end sequence at the end of each line. Depending on the stream being written, other modifications or formatting may be applied to output lines by the operating system or the hardware; however the output data remains a single logical line.

The current *write position* in a stream (identical to the read position) is also maintained by the REXX language processor. This position is usually the end of the stream (as for example when the stream is first opened), so that data can be appended to the end of the stream. For persistent files, however, the write position can be set to the beginning of the stream to overwrite existing data by giving a value of 1 for the `start` parameter of the CHAROUT function or for the `line` parameter of the LINEOUT function. Or, the STREAM function can be used to direct sequential output to some arbitrary point in the stream.

Note: Once data has been placed in a transient character output stream (for example, a network or serial link), it is no longer accessible to REXX. Normally the actual use of the characters in the stream, by the system, is asynchronous.

The STREAM Function

The built-in STREAM function is used to determine the state of an input or output stream and to carry out specific operations, described by *stream commands*. This stream command mechanism allows REXX programs to open and close selected streams for read-only or read and write operations, to move the read and write positions within a stream, and to access the size and the date of last update, see "STREAM" on page 97.

External Data Queue

The external data queue is a list of character strings that can only be accessed by line operations. It is external to REXX programs in that other REXX programs can have access to the queue.

The external data queue therefore forms a REXX-defined channel of communication between programs. Data in the queue is arbitrary; no characters have any special meaning or effect.

Apart from the explicit REXX operations described here, no detectable change to the queue occurs during the execution of a REXX program except when control leaves the program (as, for example, when an external command or routine is called). The following are the REXX queuing operations:

- Lines may be removed from the queue using the PULL or PARSE PULL instructions. When the queue is empty, these instructions will read lines from the default character input stream (STDIN:). In this way, the external data queue may be used as a source for user input, provided that the input is read as lines with PULL or PARSE PULL. Optionally, queue items can also be removed by using the LINEIN function to address the queue as a device, such as `LINEIN('QUEUE:')`.
- Lines can be stacked at the head of the queue using the PUSH instruction.
- Lines can be added to the tail of the queue using the QUEUE instruction, or using the LINEOUT function to address the queue as a device, such as `LINEOUT('QUEUE:','string')`.
- The QUEUED function returns the number of lines currently in the queue as does the function call `LINES('QUEUE:')`.

Implementation

Usually, the dialog between a REXX program with the user takes place on a line-by-line basis and is therefore carried out with the SAY and PULL (or PARSE PULL) instructions. This technique considerably enhances the usability of many programs, as they may be converted to programmable dialogs by using the external data queue to provide the input normally typed by the user. The PARSE LINEIN instruction should only be used when it is necessary to bypass the external data queue.

When a dialog is not on a line-by-line basis, use the explicitly serial interfaces provided by the CHARIN and CHAROUT functions. These functions are especially important for input and output in transient character streams, such as keyboards, printers, or network environments.

Opening and closing of persistent streams, such as files, is largely automatic. Generally speaking, a stream is opened upon the first call of a line or character function and remains open until explicitly closed with the CHAROUT, LINEOUT or STREAM functions, or until the program ends. A stream can also be opened or closed explicitly. This can be done with the STREAM function, or through specific use of the other I/O functions. For example, invoking the LINEOUT function with just the name of a stream (and no output line) closes the named stream.

A stream opened by the CHARIN, CHAROUT, LINEIN or LINEOUT functions is open for both reading and writing. The STREAM function, however, can be used to open a stream for read-only or write-only operations.

Queue Interface

REXX provides queuing services entirely separate from the OS/2 Inter-Process Communications queues. The queues discussed here are solely for the use of REXX programs.

REXX queues are manipulated within a program by these instructions:

PUSH Stacks a string on top of the queue (LIFO).

QUEUE Adds a string to the tail of the queue (FIFO).

PULL Reads a string from the head of the queue. If the queue is empty, input is taken from the console (STDIN:).

To get the number of items remaining in the queue, use the function QUEUED.

Access to Queues

There are two kinds of queues in REXX. Both kinds are accessed and processed by name.

Session Queues

One *session queue* is automatically provided for each OS/2 session in operation. Its name is always SESSION and it is created by REXX the first time information is put on the queue by a program or procedure. All processes (programs and procedures) in a session can access the session queue. However, a given process can only access the session queue defined for its session and the session queue is not unique to any single process in the session.

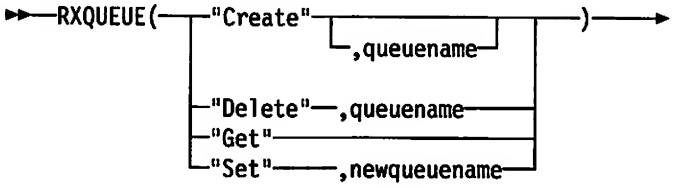
Private Queues

Private queues are created (and deleted) by your program. You can name the queue yourself or leave the naming to REXX. In order for your program to use any queue, it must know the name of the queue.

RXQUEUE Function

Use the RxQueue function in a REXX program to create and delete queues and to set and query their names. The first parameter determines the function. The entire function name must be specified but the case of the function parameter is ignored. The syntax for a valid queue name is the same as for a valid REXX symbol.

Syntax:



Parameters:

Create Creates a queue with the name, queuename (if specified); if no name is specified, then REXX will provide a name. Returns the name of the queue in either case.

Delete Deletes the named queue; returns 0 if successful, a nonzero number if an error occurs; the possible return values are:

0	Queue has been deleted.
5	Not a valid queue name.
9	Queue named does not exist.
10	Queue is busy; wait is active.
12	A memory failure has occurred.
1000	Initialization error; check file OS2.INI.

Get Returns the name of the queue currently in use.

Set Sets the name of the current queue to newqueuename and returns the previous name of the queue.

The following is an example:

```
/*                                                                  */
/*            push/pull WITHOUT multiprogramming support            */
/*                                                                  */
push date() time()          /* push date and time              */
do 1000                     /* lets pass some time             */
  nop                       /* doing nothing                   */
end                         /* end of loop                     */
pull a b                    /* pull them                       */
say 'Pushed at ' a b ', Pulled at ' date() time() /* say now and then */

/*                                                                  */
/*             push/pull WITH multiprogramming support              */
/*           (no error recovery, or unsupported env tests           */
/*                                                                  */
newq = RXQUEUE('Create')    /* create a unique queue           */
oq = RXQUEUE('Set',newq)    /* establish new queue             */
push date() time()          /* push date and time              */
do 1000                     /* lets spend some time            */
  nop                       /* doing nothing                   */
end                         /* end of loop                     */
pull a b                    /* get pushed info                 */
say 'Pushed at ' a b ', Pulled at ' date() time() /* tell user       */
call RXQUEUE 'Delete',newq         /* destroy unique queue created    */
call RXQUEUE 'Set',oq         /* reset to default queue (not required)*/
```

Figure 3. Sample REXX Procedure Using a Queue

Special Considerations

1. The size of any single data element cannot exceed 64KB minus 64 bytes, which is the size of one full segment minus a header field.
2. External programs that must communicate with a REXX procedure by means of defined data queues can use the default queue or the session queue, or they can receive the data queue name by some interprocess communication technique. This could include: parameter passing, placement on a prearranged logical queue, or use of normal OS/2 Inter-Process Communication mechanisms (for example, pipes, shared memory, or IPC queues).
3. Named queues are available across the entire system; therefore, the names of queues must be unique within the system. If a queue named os2que exists and this function is issued:

   ```
   newqueue = RXQUEUE('Create', 'os2que')
   ```

   a new queue is created with a randomly chosen name, and *newqueue* is assigned that new name.
4. Any external program started inherits as its default queue the queue in use by the parent process.

Detached Processes

1. Detached processes will access a *detached* session queue that is unique for each detached process. Note, however, that this detached session queue is *not* the same as the session queue of the starting session.
2. REXX programs that are to be run as detached processes cannot perform any SAY instructions or any PULL or PARSE PULL instructions that involve terminal I/O. However, PULL and PARSE PULL instructions that act on a queue are permitted in detached processes.

Multi-Programming Considerations

This data queue mechanism differs from the OS/2 standard API queueing in the following ways:

1. The queue is *not* owned by a specific process and as such any process is entitled to modify the queue at any time. The operations that effect the queue are atomic, in that the resource is serialized by the subsystem such that no data integrity problems can be encountered.

   However, synchronization of requests such that two processes accessing the same queue get the data in the order it was placed on the queue is a *user* responsibility and will *not* be provided by the subsystem support code.
2. Such a queue is defined as an *element copy data queue*. No single item may exceed 64KB to 64 bytes in size.
3. A regular OS/2 IPC queue is owned (created) by a specific process. When that process terminates, the queue is destroyed. Conversely, the queues created by the `RxQueue('Create', queuename)` call will exist until *explicitly* deleted. Termination of a program or procedure that created a private queue does not force the deletion of the private queue. Any data on the queue when the process *creating* it terminates will remain on the queue until either the queue is deleted, by way of the REXX function call `RxQueue('Delete', queuename)`, or until the data is read.

Data queues *must be explicitly deleted* by some procedure or program (not necessarily the creator). Deletion of a queue with remaining items, destroys those items. If a queue is NOT deleted it will be lost and cannot be recovered accept by randomly attempting to access each queue in the defined series.

Errors During Input and Output

The REXX language offers considerable flexibility in the handling of errors during input or output. It is provided in the form of a NOTREADY condition that may be trapped by the CALL ON and SIGNAL ON instructions and further information can be elicited by the STREAM function. See Chapter 7, "Conditions and Condition Traps," for a more detailed discussion of SIGNAL ON and CALL ON.)

When an error occurs during an input or output operation, the function being called will normally continue without interruption (with, for example, a nonzero count being returned by an output function). Depending on the nature of the operation, a program has the option of raising the NOTREADY condition. The NOTREADY condition is similar to the ERROR and FAILURE conditions associated with commands in that it does not cause a terminating error if the condition is raised but is not trapped.

Once NOTREADY has been raised, the following possibilities exist:

- The NOTREADY condition is not being trapped; in this case execution continues without interruption; the NOTREADY condition remains in the OFF state.
- The NOTREADY condition is being trapped by SIGNAL ON NOTREADY; in this case, the NOTREADY condition is raised, execution of the current clause ceases immediately, and the SIGNAL takes place as usual for condition traps.
- The NOTREADY condition is being trapped by CALL ON NOTREADY; in this case the NOTREADY condition is raised, but execution of the current clause is not halted. The NOTREADY condition is put into the delayed state, and execution continues until the end of the current clause. While execution continues, input functions that refer to the same stream may return the null string and output functions may return an appropriate count, depending on the form and timing of the error. At the end of the current clause, the CALL takes place as usual for condition traps.
- The NOTREADY condition is being trapped (by CALL ON NOTREADY) but is already in the DELAY state (due to NOTREADY already having been raised); in this case execution continues, and the NOTREADY condition remains in the DELAY state.

Once the NOTREADY condition has been raised and is in DELAY state, the CONDITION function will return (for a *description* invocation) the name of the stream being processed when the stream error occurred. If the stream is a default stream and has no defined name, then the null string may be returned in this case.

The STREAM function will then usually show that the state of the stream is ERROR or NOTREADY, and additional information on the state of the stream will normally be available by way of the *description* option of the STREAM function.

Examples of Input and Output

In most circumstances, communication with a user running a REXX program will be by way of the default input and output streams. For a question and answer dialog, the recommended technique is to use the SAY and PULL instructions (using PARSE PULL if case-sensitive input is required).

More generally, though, it is necessary to write to or read from streams other than the default. For example, to copy the contents of one file to another one might use the following program:

```
/* FILECOPY.CMD */
/* This routine copies the stream or file named by  */
/* the first argument to the stream or file named   */
/* by the second, as lines.                         */
parse arg inputname, outputname

signal on notready

do forever
  call lineout outputname, linein(inputname)
end

notready:
```

As long as lines remain in the named input stream, a line is read and is then immediately written out to the named output stream. It is easy to modify this program so that it filters the lines in some way before they are written.

To illustrate how character and line operations might be mixed in a communications program, consider the following example in which a character stream is converted into lines:

```
/* COLLECTOR.CMD */
/* This routine collects characters from the stream */
/* named by the first argument until a line is      */
/* complete, and then places the line on the        */
/* external data queue.                             */
/* The second argument is the single character that */
/* identifies the end of a line.                    */
parse arg inputname, lineendchar

buffer=''      /* zero-length character accumulator */
do forever
  nextchar=charin(inputname)
  if nextchar=lineendchar then leave
  buffer=buffer||nextchar          /* add to buffer */
  end
queue buffer /* place it on the external data queue */
```

Here each line is built up in a variable called BUFFER. When the line is complete (for example, when the Enter key is pressed) the loop is ended and the contents of BUFFER are placed on the external data queue. The program then ends.

Summary of Instructions and Functions

CHARIN	Reads zero or more characters from a character input stream. A start position may be specified for persistent streams. See "CHARIN" on page 76.
CHAROUT	Writes zero or more characters to a character output stream. A start position may be specified for persistent streams. See "CHAROUT" on page 77.
CHARS	Returns the number of characters currently remaining in a character input stream. See "CHARS" on page 78.
LINEIN	Reads zero or one line from a character input stream. See "LINEIN" on page 89.
LINEOUT	Writes zero or one line to a character output stream. See "LINEOUT" on page 90.
LINES	Returns 1 if any data currently remains in a character input stream. See "LINES" on page 92.
PARSE LINEIN	Reads one line from the default character input stream. See "PARSE LINEIN" on page 47.
PARSE PULL	Reads one line from the external data queue. If the queue is empty it reads a line from the default character input stream instead. See "PARSE PULL" on page 47.

PULL	The same as PARSE PULL except that the string read is translated to uppercase. See "PULL" on page 51.
PUSH	Writes one line to the head of the external data queue, as in a stack. See "PUSH" on page 52.
QUEUE	Writes one line to the tail of the external data queue. See "QUEUE" on page 53.
QUEUED	Returns the number of lines currently available in the external data queue. See "QUEUED" on page 94.
SAY	Writes one line to the default character output stream. See "SAY" on page 55.
STREAM	Returns a string describing the state of, or the result of an operation upon, a named character stream. See "STREAM" on page 97.

Chapter 9. Application Programming Interface

This chapter is addressed mainly to professional systems and application programmers. It describes:

- Data types and structures
- Invoking the REXX interpreter
- Subcommand processing
- External functions
- Macrospace interface
- Variable pool interface
- System exits.

In this chapter, the term *application* is used to refer to programs written in languages other than REXX. The features described here allow an application to extend many parts of the REXX language. This includes providing handlers for subcommand, external function and system exit processing.

Subcommands Are single clauses consisting of just an expression. The expression is evaluated and the result is passed as a command string to the currently *addressed* environment. Subcommands are used in the context of REXX running as a macro processor under some environment or program.

Functions Are direct extensions to the capabilities of REXX. An application (including REXX itself) can provide a function to augment the function set of REXX. Unlike subcommands, which generally correspond to the application's normal command set, functions are generally REXX specific and can only have a meaning from within REXX. They may be as simple as an encapsulation of several base REXX instructions or as complex as a large library of functions external to REXX written in some other programming language.

System Exits Allow REXX to operate under programmer-defined variations of the operating system. By using these exits, specific REXX actions can be executed as calls to routines provided by the caller of the interpreter, instead of using the code of the interpreter.

All of these handlers are similar in the ways that they must be coded, compiled and packaged.

In addition, REXX also provides the means for applications to manipulate the variables in REXX programs (the Variable Pool Interface), and to store and execute REXX routines in memory and thereby minimize disk access (the Macrospace Interface).

Note: Examples of calling conventions are given in the C programming language. The functions described are, however, available to all languages provided by the Common Programming Interface defined by the SAA specification. A list of supported languages is in Chapter 1.

© Copyright IBM Corp. 1983, 1990

General Characteristics

Here are some basic requirements for applications to be used as handlers for subcommand, external function, and system exit processing.

- A handler must be written as a large model program. In addition, it must:
  - Use its caller's stack segment
  - Set up its own data segment
  - Be entered by a far call
  - Follow the PASCAL calling conventions.

  Note: Under the IBM C/2 compiler version 1.1 the -Alfu compiler switches force the registers to be properly set up.
- A handler must be *packaged*, either as:
  - Part of an OS/2 Dynamic Link Library (a *dynalink* or DLL)

    or
  - Part of an executable (EXE) module.
- A handler must be *registered* with REXX before it can be used. The registration tells REXX where the handler is and how it can be invoked. For example, the information provided on the registration of a dynalink external function includes the name of the function, the name of the dynalink library containing the handler, and the name of the procedure within the dynalink that implements the function. Also note:
  - Dynalink handlers are global to the OS/2 program; they can be invoked from a REXX procedure running in any process in any session in the OS/2 program.
  - EXE handlers are local to the registering process; this means that handlers packaged as part of an EXE module can only be invoked by a REXX procedure running in the same process that registered it.

Data Types and Structures

Many of the interfaces described in this chapter require a method by which arguments can be passed. The data structure named **RXSTRING** is used for communication between the REXX interpreter and application programs. The arguments passed may consist of short strings (strings residing entirely within one segment) or huge strings (single strings spanning multiple segments).

The following figure contains a structure and macros that are used to describe a **RXSTRING**:

```
typedef struct {
   ULONG          strlength;          /*   length of string          */
   PCH            strptr;             /*   far pointer to string     */
   } RXSTRING;

typedef RXSTRING FAR *PRXSTRING;       /* pointer to a RXSTRING       */


#define RXNULLSTRING(r)         (!(r).strptr)
#define RXZEROLENSTRING(r)      ((r).strptr && !(r).strlength)
#define RXVALIDSTRING(r)        ((r).strptr && (r).strlength)
#define RXSTRLEN(r)             (RXNULLSTRING(r) ? 0 : (r).strlength)
#define RXSTRPTR(r)             (r).strptr
#define MAKERXSTRING(r,p,l)     (r).strptr=(PCH)p; (r).strlength=(ULONG)l


typedef struct {
   PSZ    sysexit_name;                /* subcommand for this sysexit*/
   SHORT  sysexit_code;                /* sysexit function code      */
} RXSYSEXIT;
```

Figure 4. REXXSAA Global Data Structures

Notes:

1. REXX arguments may either be given a value (which may be null, '') or not specified at all.
   - If an argument is specified, then the appropriate parts of **RXSTRING** will be given values.
   - If a null character ('') is specified as the argument, then *strlength* will be zero, and *strptr* will be nonzero.
   - If no argument is specified, then *strptr* will be zero.
2. The huge mode of **RXSTRING** contains exactly the same information as the short mode, except that the memory is allocated by way of a call to DosAllocHuge. Through this convention, strings in a user's program can span multiple segments.
3. In certain instances (such as, during subcommand calls, function calls, and some exit calls), the interpreter will provide **RXSTRING** with a *strlength* and buffer size of 250 for the return of information. If the data for return from these calls is less than 250 characters in length, the program should copy the data into this buffer and reset the *strlength* field accordingly.

Invoking the REXX Interpreter

The REXX interpreter can be called directly from the operating system or from within an application program.

From the OS/2 Program

The standard OS/2 CMD.EXE shell calls the interpreter for the user:

- At OS/2 command prompts
- In calls from CMD (batch) files

  Note: Use the OS/2 CALL command to invoke a REXX program in a batch file if you want control to return to the caller.

- From a start programs group window in Presentation Manager.

When a REXX program is run, the source is first converted into an internal form. The program is run using this internal form. The internal form is saved on disk in the extended attribute area of the source file. Thus, it is available to use the next time the program is run. This increases the performance of REXX programs.

The maximum size of the internal form that can be saved is about 64K bytes. If you have a program that is larger than this, the internal form is created and discarded each time you run the program. You can increase the performance of the program by breaking it up into smaller programs.

You may occasionally want to create and save the internal form of a program without running it. To do this, type the name of the program at the OS/2 prompt and follow it with the parameter "//T".

The special string "//" is reserved for use by REXX to mark parameters to the REXX interpreter. If it is found on a command that runs a REXX program, it is not passed to the program.

From within an Application

The REXX processor is an OS/2 dynamic link library (DLL) that is fully re-entrant and allows recursive entry by the same thread in a process and supports different procedures running independently in multiple threads within the same process.

A C-language prototype for calling REXX is included in the file REXXSAA.H in the installation package.

Interface Functions

This is the function that allows calls to the REXX interpreter from an application program. Procedures run in the same session, process, and thread as the caller.

REXXSAA

This call allows the user to call the REXX interpreter from an application program.

REXXSAA (ArgCount, ArgList, ProgramName, Instore, EnvName, CallType, Exits, ReturnCode, Result)

Parameters

ArgCount *(SHORT) – input*
Is the number of arguments supplied for this call to the REXX interpreter.

ArgList *(PRXSTRING) – input*
Is an array of **RXSTRING** items that describes the arguments for this call to the REXX interpreter.

ProgramName *(PSZ) – input*
Is a pointer to an ASCIIZ string containing at least the filename and optionally the extension and full drive and path specification of the REXX procedure. If no file extension is specified, then a default of .CMD is supplied. If no path is given, then the normal OS/2 file search (current directory then environment path) will be performed, for example, C:\UTIL\PLAYTIME.CMD.

Instore *(PRXSTRING) – input*
Is an array of two descriptors for instorage REXX procedures:

Instore[0] An **RXSTRING** item describing the memory buffer containing the source of the procedure. The source must be in the same format as if it were on disk that is, complete with carriage returns, line feeds, and end-of-file characters implied at end-of-string.

Instore[1] A second **RXSTRING** item used initially to return the tokenized image of the source in DosAllocSeg or DosAllocHuge space from the interpreter to the caller. Subsequently, the caller can pass this tokenized image back to the interpreter. The interpreter will use this tokenized image and thus save the pre-tokenization time. See the following remarks.

If **Instore** is specified, then no disk search will be performed; *ProgramName* will only be used as the name of the procedure. If no instorage buffer is to be supplied, then the *Instore pointer* should be null.

EnvName *(PSZ) – input*
Is a pointer to a string specifying the initial interpreter address environment. If an environment name is not specified, then the initial address defaults to the file extension. If specified, this value must be in the form of an ASCIIZ string.

CallType *(SHORT) – input*
Is a flag describing how REXX was called; possible values are:

RXCOMMAND	as a command
RXSUBROUTINE	as a subroutine
RXFUNCTION	as a function

Exits *(PRXSYSEXIT) – input*
Is an array of **RXSYSEXIT** elements that contains pointers to subcommand environment names and an integer code identifying the exit. The last **RXSYSEXIT** element must contain the **RXENDLST** code that indicates the end of the system exit list. Set this parameter to null if there are no system exits provided. See "System Exits" on page 191 for system exit details.

ReturnCode *(PSHORT) – output*
If the **Result** string is numeric, then it will be converted to an integer and also returned in **ReturnCode**.

Result *(PRXSTRING) – output*
Is the **RETURN** or **EXIT** value from the *ProgramName* program. The returned string should be released by a call to DosFreeSeg.

Returns

An integer return code from a call to REXXSAA may be issued by the REXX interpreter or from the OS/2 program. The possible return codes are:

negative	Interpreter errors. These are the standard SAA-defined interpreter error codes. Refer to Appendix A, "Error Numbers and Messages," for a description of these error conditions.
0	No errors occurred. Procedure is executed normally.
positive	OS/2 return code indicating problems finding or loading the interpreter. See the return codes for the OS/2 functions DosLoadModule and DosGetProcAddr for details.

Remarks

With regard to instorage procedures:

- Memory allocated by **Instore** must be released by way of DosFreeSeg by the caller; once the tokenized image is returned to the caller the memory is no longer tracked by REXX.
- Whenever the source is changed, the tokenized image must be released and **Instore[1]** must specify an empty **RXSTRING** item.
- If **Instore[1]** is specified, then **Instore[0]** need not actually contain the source (that is, it may be empty).
- To prevent instorage tokenization, set **Instore[1]** to an invalid **RXSTRING** item.

The following is an example for calling the REXX interpreter from an application program.

```
short     argc, rc;
char      *rexxsaafile, *envnam;
RXSTRING  *argv, instor, retval;
RXSYSEXIT *sysexit;

return_code = REXXSAA(argc, argv, rexxsaafile, instor, envnam, RXCOMMAND, sysexit, &rc, &retstr);
```

Figure 5. Sample Call to the REXX Interpreter

Subcommand Interface

An application can dynamically be made known by name to the REXX interpreter as an environment accessible by the ADDRESS instruction (see page 24). To do this, the application must be registered as a specific subcommand environment. This can be accomplished through the RXSUBCOM interface.

Data Definitions

In order to register a subcommand environment, a subcommand request block or **SCBLOCK** must be created. The following is the format of **SCBLOCK**:

```
/***    Structure of REXX Subcommand Block (SCBLOCK) */

typedef struct subcom_node {
    struct subcom_node far *next;      /* Pointer to the next block   */
    PSZ    scbname;                    /* Subcom environment name     */
    PSZ    scbdll_name;                /* Subcom module name          */
    PSZ    scbdll_proc;                /* Subcom procedure name       */
    double scbuser;                    /* User area                   */
    PFN    scbaddr;                    /* Subcom environment address  */
    USHORT scbmod_handle;              /* Dynalink module handle      */
    USHORT scbdrop_auth;               /* Permission to drop          */
    PID    scbpid;                     /* Process ID of registrant    */
    USHORT scbsid;                     /* Session ID.                 */
    } SCBLOCK;

typedef SCBLOCK FAR *PSCBLOCK;
```

Figure 6. Subcommand Data Definitions

Within the **SCBLOCK** you must specify either:

- **For dynalink registration:** the subcommand environment name (scbname), the library name (scbdll_name), and the procedure name (scbdll_proc)

  or

- **For EXE-module registration:** the subcommand environment name (scbname) and the handler address (scbaddr).

Parameters

scbname Is an ASCIIZ string to be registered as the subcommand environment name.

scbdll_name Is an ASCIIZ string specifying the library name of the dynalink library containing the subcommand handler. **scbdll_name** is the field that REXX uses to determine if this is a .DLL or .EXE registration. If **scbdll_name** is a null pointer (0), then this is assumed to be an .EXE registration. If **scbdll_name** is non-null, then this is assumed to be a DLL registration. The library name is the name supplied by the module definition file for this library.

scbdll_proc Is an ASCIIZ string specifying the procedure name of the subcommand handler. Used by DosGetProcAddr to get the address of the subcommand handler procedure.

scbaddr The actual address of the subcommand environment handler. This parameter is used by .EXE programs supplying an internal routine as the subcommand handler. These handlers are *local* to the current process (that is, only subcommands originating in the same process as the registration process of *local* handlers will be presented to the handler).

The following fields are optional:

scbdrop_auth Drop authority. Determines which processes can drop the subcommand handler; the options are:

RXSUBCOM_DROPPABLE
Allows any process to drop this subcommand handler using the RxSubcomDrop interface, see page 165. This is the default setting.

RXSUBCOM_NONDROP
Only a thread of execution with the same session ID and process ID as the thread that registered this handler will be allowed to complete a RxSubcomDrop request against this handler.

scbuser User area to be saved with subcommand registration information.

Writing Subcommand Handlers

Because subcommands are closely associated with a given application, they are written as part of the application. The subcommand code can reside in the same module (.EXE or .DLL) as the application, or can reside in an accessory dynalink. Any separate procedure that can be registered with REXX and follows the appropriate conventions can be a handler.

For details on the requirements and options for packaging API handlers, see "General Characteristics" on page 152. As documented in the OS/2 Technical Reference, the *library name* must be the same as the file name of the dynalink or .EXE files being used.

Documentation of subcommand handlers packaged as dynalink libraries must specify the *library name* (used by DosLoadModule) of the dynalink created; see note 1 on page 160. This *library name* must be used in order to register the subcommand and may be needed in order to resolve duplicate environment names. The procedure name of the subcommand handler must also be published. This procedure name must be exported from the dynalink library.

Invocation

The following is a sample subcommand handler definition and invocation.

```
SHORT APIENTRY my_subcom (
        RXSTRING,       /* Command string passed from the caller    */
        PUSHORT,        /* pointer to short for return of flags     */
        PRXSTRING       /* pointer to RXSTRING for return string    */
        );

unsigned short pascal far my_subcom (command, flags, retstr)
RXSTRING command;
unsigned short far *flags;
RXSTRING far *retstr;
```

Figure 7. Sample Definition and Invocation of a Subcommand Processor

Where:

command Is the entire command string as resolved by REXX.

flags Are flags returned by the subcommand handler to the interpreter to indicate successful completion, error, or failure conditions.

RXSUBCOM_OK	Subcommand completed normally
RXSUBCOM_ERROR	Error in subcommand
RXSUBCOM_FAILURE	Failure in subcommand.

retstr The address of a **RXSTRING** to define your result in. See "Returning Result Values" on page 169 for additional information.

These arguments correspond to the same arguments defined for the function RxSubcomExecute, described on page 163:

- The subcommand handler is passed the command as **RXSTRING**, the pointer to the flag storage area and the pointer to **RXSTRING** for the subcommand result string.
- The subcommand returns an integer return code that is placed in the REXX special variable **RC** or returned to the caller of RxSubcomExecute in the last (**RC**) argument:
  - If a nonzero number is returned, that number is used as the return code.
  - If zero is returned and the result **RXSTRING** is not empty, then the return code is set to the value of the result **RXSTRING**.
  - Otherwise the return code is set to zero.

Duplicate Environment Names

Multiple subcommand handlers can register themselves for any given subcommand environment name. However, for any given OS/2 program startup, there are unique *library names* associated with individual handlers packaged in DLLs see note 1. These *library names* should be documented and published with the instructions for the subcommand handlers. In order to make subcommand environment names unique, the user can incorporate this documented *library name* into the subcommand environment addressed.

For example:

- QED Corp. produces an editor whose subcommand environment is EDIT that is packaged in QEDIT.DLL and has a library name of QEDIT.
- XYZ Corp. produces a word processor whose subcommand environment is also EDIT but is packaged as XWORD.DLL and has a library name of XWORD.

The subcommand handler to be used can be identified by providing the library name when addressing the subcommand environment, for example:

- The subcommand `ADDRESS 'Edit.Qedit'` sends subcommands to the EDIT environment established by the QEDIT dynalink.
- The subcommand `ADDRESS 'Edit.Xword'` sends subcommands to the EDIT environment established by the XWORD dynalink.

Notes:

1. The *library names* mentioned here are the ModuleNames used in DosLoadModule calls. These names are found in the OS/2 LIBPATH path so only one handler of *library name* can be found until LIBPATH is changed and the system restarted or libraries are unloaded and moved around in the subdirectories searched by way of LIBPATH.
2. Providing library qualifying information is only necessary for duplicate dynalink subcommand environments because of the established subcommand environment search order.

Search Order

Since multiple subcommand handlers can register using the same environment name, it is necessary to establish a subcommand handler search order.

When a dynalink library name is specified with the ADDRESS instruction, REXX invokes the handler registered for that environment and packaged in that library. If the specified handler is not available, then an error is returned.

If a dynalink library name is not specified then:

- The registered handler chain is searched for a matching environment name that was registered from the current process ID and session ID (essentially an .EXE subcommand handler entrypoint).
- If there is no local handler, then search the registered handler chain for the first matching handler that can be found and used.
- If none is found, then an error is returned.

Interface Functions

The following are the functions for registering and manipulating subcommand interfaces.

RxSubcomRegister

This call allows the user to register a subcommand environment.

RxSubcomRegister (SubComBlock)

Parameters

SubComBlock *(PSCBLOCK) – input*
Is a pointer to a user-allocated **SCBLOCK**, see "Data Definitions" on page 157.

Returns

0	RXSUBCOM_OK
10	RXSUBCOM_DUP
1002	RXSUBCOM_NOEMEM
1003	RXSUBCOM_BADTYPE.

RxSubcomQuery

This call allows the user to query a subcommand environment name.

RxSubcomQuery (EnvName, ModuleName, Flag, UserWord)

Parameters

EnvName *(PSZ) – input*
Is a pointer to a null-terminated (ASCIIZ) subcommand environment name string.

ModuleName *(PSZ) – input*
Is a pointer to a null-terminated (ASCIIZ) string containing the library name of the dynalink library that contains the registered subcommand handler. This name can be NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, then the search order documented in "Search Order" on page 160 is used for environment resolution. Also see "Duplicate Environment Names" on page 160.

Flag *(PUSHORT) – output*
Is an indicator of whether the subcommand identified by **EnvName** is registered or not. If **Flag** is set to 0, then the subcommand is not registered. If **Flag** is set to 1, then the subcommand is registered.

UserWord *(double far *) – output*
Is the double word of data saved as *scbuser* during the registration process.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
1003	RXSUBCOM_BADTYPE.

Remarks

The caller may query the status of a particular environment name. The caller passes a pointer to the name of an environment. A flag indicating the existence of the subcommand (1 = registered, 0 = not registered) is returned as well as the userword saved at registration.

RxSubcomExecute

This call allows the user to send commands to a subcommand environment.

RxSubcomExecute (EnvName, ModuleName, Command, Flags, RetVal, Result)

Parameters

EnvName *(PSZ) – input*
Is a pointer to a null-terminated (ASCIIZ) subcommand environment name string.

ModuleName. *(PSZ) – input*
Is a pointer to a null-terminated (ASCIIZ) string containing the library name of the dynalink library that contains the registered subcommand handler. This name can be NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, then the search order documented in "Search Order" on page 160 is used for environment resolution. Also see "Duplicate Environment Names" on page 160.

Command *(RXSTRING) – input*
Is a string describing the entire command string.

Flags *(PUSHORT) – output*
Is a pointer to an integer of flags, which is used to notify the caller of failure to locate a subcommand or of errors encountered while the subcommand is executed. This information is reflected by the interpreter during **TRACE** (errors) activity. See RxSubcom Flag Definitions on page 159 for definitions of the flags.

RetVal *(PUSHORT) – output*
Is the return code from the subcommand handler.

Result *(PRXSTRING) – output*
Is the result of the subcommand processing. The subcommand handler modifies the **Result** structure for the returned string.

Note: The string returned from the subcommand handler is allocated segments that are given to the caller. The caller uses DosFreeSeg to free the returned string when finished with it, see "Returning Result Values" on page 169.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
50	RXSUBCOM_LOADERR
127	RXSUBCOM_NOPROC.

Remarks

The REXX interpreter invokes the appropriate subcommand handler for the specified subcommand environment name and library (if needed).

RxSubcomLoad

This call allows the user to load a subcommand environment packaged in a dynalink.

RxSubcomLoad (EnvName, ModuleName)

Parameters

EnvName *(PSZ) – input*
Is a pointer to a null-terminated (ASCIIZ) subcommand environment name string.

ModuleName *(PSZ) – input*
Is a pointer to a null-terminated (ASCIIZ) string containing the library name of the dynalink library that contains the registered subcommand handler. This name can be NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, then the search order documented in "Search Order" on page 160 is used for environment resolution. Also see "Duplicate Environment Names" on page 160.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
50	RXSUBCOM_LOADERR
127	RXSUBCOM_NOPROC.

Remarks

If a subcommand environment is registered by library and routine name, then a caller may wish to have the REXX interpreter issue an explicit load for the environment-handling routines.

RxSubcomDrop

This call allows the user to drop a subcommand environment.

RxSubcomDrop (EnvName, ModuleName)

Parameters

EnvName *(PSZ) – input*

Is a pointer to a null-terminated (ASCIIZ) subcommand environment name string.

ModuleName *(PSZ) – input*

Is a pointer to a null-terminated (ASCIIZ) string containing the library name of the dynalink library that contains the registered subcommand handler. This name can be NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, then the search order documented in "Search Order" on page 160 is used for environment resolution. Also see "Duplicate Environment Names" on page 160.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
40	RXSUBCOM_NOCANDROP
1003	RXSUBCOM_BADTYPE.

Remarks

The environment is dropped or deregistered from the active subcommand environment list.

Return Codes

RXSUBCOM_ISREG	0x01	The subcommand environment has been registered.
RXSUBCOM_ERROR	0x01	An error in execution has occurred; the interpreter raises an ERROR condition.
RXSUBCOM_FAILURE	0x02	A failure in execution has occurred; the interpreter raises a FAILURE condition.
RXSUBCOM_NOEMEM	1002	There is insufficient memory available to complete this request.
RXSUBCOM_BADTYPE	1003	Bad registration type; an internal failure may have occurred. Retry the operation; if this error persists, contact your IBM representative.
RXSUBCOM_OK	0	The subcommand has been successfully executed.
RXSUBCOM_DUP	10	A duplicate environment name has been successfully registered; there is either: • An .EXE environment by the same name registered in another process or • A DLL environment by the same name registered in another DLL; to address this subcommand, its library name must be specified.
RXSUBCOM_MAXREG	20	Cannot register anymore blocks.
RXSUBCOM_NOTREG	30	This indicates either: • Registration was unsuccessful due to duplicate environment and dynalink names (RxSubcom Register) or • The subroutine environment is not registered (other RxSubcom functions).
RXSUBCOM_NOCANDROP	40	The subroutine has been registered as *not droppable*.
RXSUBCOM_LOADERR	50	An error has occurred while loading a routine into memory; most commonly this is due to a missing file.
RXSUBCOM_NOPROC	127	The RxSubcom routine was not found; check for a missing file.

```
char far *name, far *dllname;
unsigned short exist, flags, rc, return_code;
double   userword;
SCBLOCK  scb;
RXSTRING command, retstr;

return_code = RxSubcomRegister (&scb);
return_code = RxSubcomDrop     (name, dllname);
return_code = RxSubcomLoad     (name, dllname);
return_code = RxSubcomQuery    (name, dllname, &exist, &userword);
return_code = RxSubcomExecute  (name, dllname, command, &flags, &rc, &retstr);
```

Figure 8. Sample function calls of RXSUBCOM services

External Functions

There are two types of external functions:

1. Routines written in REXX
2. Routines written in other OS/2 program-supported languages.

External functions written in the REXX language are not registered with REXX; they are found by a disk search for a file name that matches the function name being called. Functions written in other languages, however, must be registered. Whether they are packaged as part of an EXE module or as a procedure within a dynalink library, REXX must be told where they are so that the correct invocation mechanism can be used.

Note: Both kinds of external functions are subject to a standard search order for all subroutines and functions; this is discussed in "Search Order" on page 66.

Registering Function Packages

The parameters needed for each kind of registration are described here. The actual interfaces for registration and execution are described on page 170.

For details on the general requirements and options for packaging API handlers, see "General Characteristics" on page 152.

Dynamic Link Library Functions

A dynamic link library file has a name and one or more externally callable entry points. The name and entry points are defined both in the function source code and in the .DEF file provided to the linker.

Functions available in DLLs are registered by:

REXX-visible name — An ASCIIZ-string name that is called within a REXX program; for example,

```
Say MyBeep(200,200)
```

Dynamic Link Library — An ASCIIZ-string name that identifies the DLL file containing the requested function; for example,

```
MYDLL
```

The DLL file must exist in a directory included in the LIBPATH system-environment variable, which is set by CONFIG.SYS when the system is started.

Entry point

An ASCIIZ-string name that identifies the visible or available entry point into the DLL used to access the proper function; for example,

```
USHORT APIENTRY MYENTRYPOINT(...)
```

If multiple functions are to be accessed through the same entry point, then the proper function name is determined from the first parameter of RxFunctionCall.

Executable Functions In Memory

Functions that are part of executable files loaded in memory are registered by:

REXX-visible name

An ASCIIZ-string name that is called within a REXX program; for example,

```
Say MyBeep(200,200)
```

Transfer address

The address of a function in memory that accepts the proper parameters and performs the required operations for the function call; for example,

```
(PSZ)(&MyFunction())
```

Writing External Functions

The following is the format for writing an external function:

```
SHORT APIENTRY my_rexx_function (PSZ, SHORT, PRXSTRING, PSZ, PRXSTRING);

short far pascal my_rexx_function (name, argc, argv, queue, retstr);
char         * name;            /* name the function is invoked as     */
short          argc;            /* number of arguments                 */
RXSTRING far   argv[];          /* argument array                      */
char     far * queue;           /* name of current queue               */
RXSTRING far * retstr;          /* RXSTRING for function result        */
```

Figure 9. Sample External Function Definition

Where:

name Is the name by which the function is invoked.
argc Is the size of the argument list, namely the number of elements passed.
argv Is an array of RXSTRING commands describing the arguments.
queue Is the name of the queue used by the function.
retstr Is the result RXSTRING; see "Returning Result Values" on page 169.

Functions must return a valid RXSTRING for the *retstr* parameter. If there is no value to return from the function, an empty RXSTRING can be used. If called as a function and no value is returned, the interpreter will generate error 44, `Function did not return data`.

For an RXSTRING in huge form, the storage area must be created by way of a call to DosAllocHuge(). For RXSTRING in short form, the storage area must be

created by way of a call to DosAllocSeg(). The interpreter will destroy any RXSTRING passed to it as a return value from a function call. Any program that wishes to maintain the information passed to REXX must create its own copy before transferring control back to the interpreter.

Returning Result Values

Subcommands and external functions are capable of returning **RXSTRING** result strings that are returned in the special variable **RC** for subcommands or in **RESULT** for external subroutines (external function results are processed in line to the calling clause). In order to accomplish this, function and subcommand handlers are passed an address to a **RXSTRING** of length 250 in the REXX interpreter's storage space. The processor can use this **RXSTRING** or return a far pointer to memory allocated by DosAllocSeg or DosAllocHuge. If this memory is allocated by a process other than the one in which the interpreter is running (for example, during execution of another program called by the function), it must be made available to the interpreter by way of a call to DosGiveSeg. The interpreter frees this storage after copying the result string into the appropriate interpreter storage.

The possible **RXSTRING** result values are:

1. A null *strptr* (for example, *strptr* = zero) is assumed to represent no result returned.
2. A *strlength* of zero with a valid *strptr* is assumed to represent a zero-length result returned.
3. A nonzero *strlength* with a valid *strptr* is assumed to represent a valid result returned.

Calling External Functions

Functions registered with REXX by address can only be used in a program running in the same process, that is, one having the same process identification. It is possible to have different functions registered with the same REXX-visible name, as long as they are all registered by address and are all registered from different processes. If, however, a function is registered as part of a dynamic link library, then it is a global function, and its REXX-visible name cannot be repeated.

Interface Functions

The following are the functions for registering and manipulating function packages.

RxFunctionRegister

This call allows the user to register a function either as a dynalink procedure or as an address in a program in memory.

RxFunctionRegister (FuncName, ModuleName, EntryPoint, ModuleType)

Parameters

FuncName. *(PSZ) – input*
Is the ASCIIZ string containing the name of the external function.

ModuleName *(PSZ) – input*
Is the ASCIIZ string containing the name of the dynamic link library for the function.

EntryPoint *(PSZ) – input*
Is the ASCIIZ string containing the name of the dynalink procedure to perform the function. (See Figure 9 on page 168 for a sample definition of a function.)

ModuleType *(USHORT) – input*
Is a flag indicating in what type of module the function resides. The valid registration-type identifiers are:

RXFUNC_DYNALINK
Indicates that the function is available in a dynamic link library.

RXFUNC_CALLENTRY
Indicates that the function is registered as an entry point in memory.

Returns

0	RXFUNC_OK
10	RXFUNC_DEFINED
20	RXFUNC_NOMEM.

Remarks

When **ModuleType** is **RXFUNC_CALLENTRY**, then a function address in the current program is registered. Since the third parameter is defined as a PSZ, the address of the function must be cast on the call. The **ModuleName** parameter is ignored for this type of registration.

RxFunctionCall

This call allows the user to call a function registered with REXX as an external function.

RxFunctionCall (FuncName, ArgCount, ArgList, RetVal, Result, QueueName)

Parameters

FuncName *(PSZ) – input*
Is the ASCIIZ string containing the name of the external function to call.

ArgCount *(USHORT) – input*
Is the number of arguments being passed to the function.

ArgList *(PRXSTRING) – input*
Is a pointer to the list of arguments being passed to the function.

RetVal *(PUSHORT) – output*
Is the address of a place for the function to put a return code.

Result *(PRXSTRING) – output*
Is the descriptor for the result string from the function. (See "Returning Result Values" on page 169.)

QueueName *(PSZ) – input*
Is the default queue for use by the function.

Returns

0	RXFUNC_OK
30	RXFUNC_NOTREG
40	RXFUNC_MODNOTFND
50	RXFUNC_ENTNOTFND.

Remarks

The function invoked will be the most recent registration of **FuncName**, with certain restrictions. If a function is found with a registered name of **FuncName**, but is not available to the current process, then the search for the function continues as if no registration had been found.

A function registered with the **RXFUNC_DYNALINK** flag is executed if it is the first valid registration found.

RxFunctionDeregister

This call allows the user to remove a function.

RxFunctionDeregister (FuncName)

Parameters

FuncName *(PSZ) – input*
Is the ASCIIZ string containing the name of the external function to remove.

Returns

0	RXFUNC_OK
30	RXFUNC_NOTREG.

RxFunctionQuery

This call allows the user to query the list of available functions for a particular function.

RxFunctionQuery (FuncName)

Parameters

FuncName *(PSZ) – input*
Is the ASCIIZ string containing the name of the external function to search for.

Returns

0	RXFUNC_OK
30	RXFUNC_NOTREG.

Remarks

This function will only return RXFUNC_OK if the requested function is available for calling by the current process. If a function is found with a registered name of **FuncName**, but is tagged as belonging to a different process, then the search for the function continues as if no registration had been found.

The following is an example for registering and manipulating function packages.

```
#define NULL_PTR 0L
char     *func, *dll, *proc, *queue;
ushort   argc;
short    rc, return_code;
RXSTRING *argv, retstr;

return_code = RxFunctionRegister   (func, dll, proc, RX_DYNALINK);
return_code = RxFunctionRegister   (func, NULL_PTR, (PSZ) &my_rexx_function, RX_CALLENTRY);
return_code = RxFunctionDeregister (func);
return_code = RxFunctionQuery      (func);
return_code = RxFunctionCall       (func, argc, argv, queue, &rc, &retstr);
```

Figure 10. External Function APIs - Sample Calls

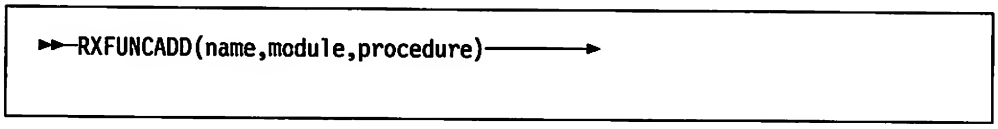
Return Codes

RXFUNC_OK	0	The call to the API completed successfully.
RXFUNC_DEFINED	10	The requested function is already defined in the table.
RXFUNC_NOMEM	20	There is not enough memory to add a new function to the table.
RXFUNC_NOTREG	30	The requested function is not registered in the table.
RXFUNC_MODNOTFND	40	The dynamic link library module could not be found.
RXFUNC_ENTNOTFND	50	The function entry point could not be found.

REXX Interface

The following built-in REXX functions can be used in a REXX procedure to register, drop, or query external functions.

RXFUNCADD

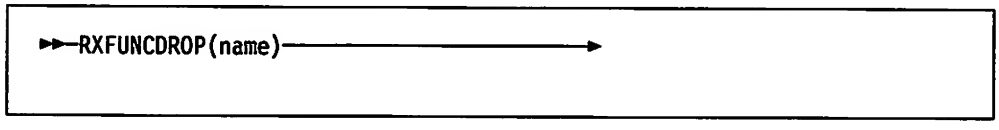


RXFUNCADD registers the function name, making it available to REXX procedures. The function will be found in the dynamic link library module, must be in a directory on LIBPATH, with transfer point procedure in the library. RXFUNCADD returns 0 if the function is registered successfully; 1 otherwise.

Notes:

1. If module, procedure, or both do not exist in the library, registration will still succeed, but an error will occur on invocation of the function.
2. Note that only dynamic link library functions can be registered from within a REXX procedure.

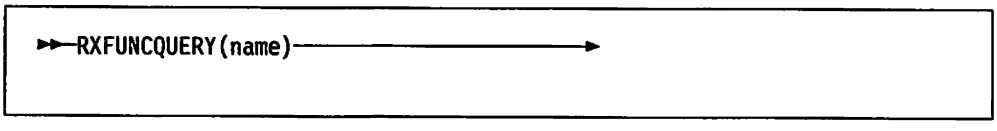
RXFUNCDROP



RXFUNCDROP removes (deregisters) the function name from the list of functions available to the current process. RXFUNCDROP returns a value of 0 if the function is dropped successfully, a value of 1 if it is not.

Note: Any process can issue a drop function regardless of which process requested the registration of the function. Therefore special care must be taken when issuing a drop request. It is not necessary to drop a function from the table, unless another function is to be used with the same REXX-usable name in another program.

RXFUNCQUERY



RXFUNCQUERY queries the list of available functions for the availability of the function name. RXFUNCQUERY returns a value of 0 if the function is available, a value of 1 if it is not.

Macrospace Interface

The macrospace interface can reduce the search time for REXX functions existing on disk by allowing a function image to be maintained in memory for immediate load and execution. This is especially useful for procedures and functions which are used frequently in a program, such as editor macros and other high-use functions.

As far as the REXX programmer is concerned, calling a macrospace-registered function is essentially identical to calling any other function. The macrospace functions are global to all REXX procedures. The sole difference in execution is the priority it is given in the external-function search order. Functions registered in the macrospace can be placed in this search order to be executed before or after all other external functions.

The REXX interpreter environment during the execution of macrospace functions is the same as if the functions existed in external REXX files.

Search Order

When a function is registered in the macrospace by way of the interface functions, a flag is passed requesting the function position in the external function search order for the REXX interpreter. There are only two valid requests for search order positioning; before all other external functions and after all other external functions. (For a diagram of the search pattern, see Figure 2 on page 68.

Pre-External Function Registration

If a function is registered with the **RXMACRO_SEARCH_BEFORE** flag, then that function will be located by the REXX interpreter *before* any functions existing in external REXX files or in function packages. This allows the user to temporarily supersede the normal execution of a function by replacing an external function without deleting or renaming the file or canceling the function registration in function packages.

Post-External Function Registration

If a function is registered with the **RXMACRO_SEARCH_AFTER** flag, then that function will be located by the REXX interpreter *after* any functions existing in external REXX files or in function packages. This allows the user to provide a default function for use by the REXX interpreter in the event that some external function required by the interpreter cannot be found in an external REXX file or function packages.

Storage of Macrospace Libraries

Functions registered in the macrospace can be saved to a permanent storage file on disk for loading at a later date. This allows an application, such as an editor, to create its own library of frequently-used functions, and load the entire macrospace library into memory for fast access by the language processor. This macrospace library is specified by a filename. Thus, multiple macrospace function libraries can exist concurrently on a disk.

Some consideration must be given to storage and memory utilization.

The fact that the macrospace interface is designed to enhance performance for external REXX functions places severe limitations on the number of restrictions which can be imposed by the API. Therefore, the only limitations which will be placed on the user will be the constraints of available memory imposed by the OS/2

program. The macrospace will be allowed to grow so long as memory is available. However, if the macrospace grows too large (relative to the amount of available memory), then performance of the operating system may deteriorate due to an increased number of disk accesses to maintain its swap file.

It is recommended, then, that the macrospace not be used to hold all of the functions needed by the user for a normal day of use. A more efficient method is to maintain only those functions that are going to be used by the current process in the macrospace, and to either delete or omit others from the macrospace.

Interface Functions

This section contains a description of the functions used to register and manipulate functions using the macrospace interface.

RxMacroChange

This call allows the user to change a function in or add a function to the macrospace.

RxMacroChange (FuncName, SourceFile, Position)

Parameters

FuncName *(PSZ) – input*
Is the ASCIIZ string specifying the name by which the function will be known. This function name must be validated to follow the REXX rules for function names; it should also correspond to function names implemented as part of external function packages.

SourceFile *(PSZ) – input*
Is the ASCIIZ string file specification of the disk file containing the source for this function. If no file extension is supplied, it will default to .CMD. If the full path is not specified, the standard search will be conducted (search current directory and OS/2 path).

Position *(USHORT) – input*
Is the flag indicating where this function should be placed in the external function search order for REXX. There are two possible positions; a function can be placed at the beginning of the search order (that is, before all other external functions) or at the end of the search order (that is, after all other external functions).

Returns

0	RXMACRO_OK
1	RXMACRO_NO_STORAGE
7	RXMACRO_SOURCE_NOT_FOUND
8	RXMACRO_INVALID_POSITION.

RxMacroDrop

This call allows the user to delete a function from the macrospace.

RxMacroDrop (FuncName)

Parameters

FuncName *(PSZ) – input*
Is the ASCIIZ string specifying the name of the function to remove.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND.

RxMacroErase

This call allows the user to erase all functions in the macrospace.

RxMacroErase ()

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND.

Remarks

There are no parameters for this function. Special care should be taken in using this function, because no verification is done on the deleted functions, that is, all functions are removed from the macrospace, regardless of whether or not they are being used by other processes.

RxMacroSave

This call allows the user to save either all or a specified subset of all functions in the macrospace to a specified file.

RxMacroSave (FuncCount, FuncNames, MacroLibFile)

Parameters

FuncCount *(USHORT) – input*
Is the number of functions requested for saving to the file.

FuncNames *(PSZ FAR *) – input*
Is a pointer to a list of ASCIIZ strings containing the names of the functions to be saved to the file; the number of strings in the list is given by **FuncCount**.

MacroLibFile *(PSZ) – input*
Is the ASCIIZ filename string to which to save the workspace. If a file exists, it is overwritten. If a file does not exist, it is created.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND
3	RXMACRO_EXTENSION_REQUIRED
5	RXMACRO_FILE_ERROR.

Remarks

If **FuncCount** is zero or **FuncNames** is null, then all of the functions in the macro space will be saved.

The **FuncNames** parameter is a pointer to a list of pointers, similar to the *argv* parameter list in the **main()** function of a C program.

RxMacroLoad

This call allows the user to load either all or a specified subset of all functions from a specified file into the macrospace.

RxMacroLoad (FuncCount, FuncNames, MacroLibFile)

Parameters

FuncCount *(USHORT) – input*
Is the number of functions requested for loading from the source file.

FuncNames *(PSZ FAR *) – input*
Is a pointer to a list of ASCIIZ strings containing the names of the functions to be loaded from the file (number of strings in list given by **FuncCount**).

MacroLibFile *(PSZ) – input*
Is the ASCIIZ filename string from which to load the workspace.

Returns

0	RXMACRO_OK
1	RXMACRO_NO_STORAGE
2	RXMACRO_NOT_FOUND
4	RXMACRO_ALREADY_EXISTS
5	RXMACRO_FILE_ERROR
6	RXMACRO_SIGNATURE_ERROR.

Remarks

If **FuncCount** is zero or **FuncNames** is null, then all of the functions in the file will be loaded.

The **FuncNames** parameter is a pointer to a list of pointers, similar to the *argv* parameter list in the **main()** function of a C program.

If a request is made to load a function that already exists (that is, if there is already a function with the same name in the macrospace), then the entire load request will be discarded and the macrospace will remain unchanged.

RxMacroQuery

Allows the user to search for a function in the macrospace.

RxMacroQuery (FuncName, Position)

Parameters

FuncName *(PSZ) – input*
Is the ASCIIZ string specifying the name of the function to search for in the list of workspace functions.

Position *(PUSHORT) – output*
Is a pointer to a PUSHORT for return of the current search-order position of the function, if it exists. If the function does not exist in the macrospace, then the value will not be changed.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND.

RxMacroReOrder

Allows the user to change the search-order location of a function in the macrospace.

RxMacroReOrder (FuncName, Position)

Parameters

FuncName *(PSZ)* – *input*
Is the ASCIIZ string specifying the name of the function whose search order positioning is to be changed.

Position *(USHORT)* – *input*
Is the flag indicating where this function will be moved to in the external function search order for REXX.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND
8	RXMACRO_INVALID_POSITION.

Search Order Flags

The following are the flags that are passed to the macrospace interface to specify where a function should be placed in the REXX search order for external functions.

RXMACRO_SEARCH_BEFORE	Places the function at the top of the search order.
RXMACRO_SEARCH_AFTER	Places the function at the bottom of the search order.

Return Codes

The following are the return codes from the macrospace interface functions. These values signify the causes for a failure, if any, in the calls to the functions.

RXMACRO_OK	0	The call to the API completed successfully.
RXMACRO_NO_STORAGE	1	There was not enough memory to complete the requested function.
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace.
RXMACRO_EXTENSION_REQUIRED	3	An extension is required for the file name passed to the function.
RXMACRO_ALREADY_EXISTS	4	Duplicate functions cannot be loaded from the requested file.
RXMACRO_FILE_ERROR	5	An error occurred accessing the requested file.
RXMACRO_SIGNATURE_ERROR	6	The file specified as containing macrospace function images is not valid.
RXMACRO_SOURCE_NOT_FOUND	7	The requested file was not found.
RXMACRO_INVALID_POSITION	8	An invalid search-order position request flag was passed to the function.

Variable Pool Interface

The Variable Pool Interface for the OS/2 program is used by application programs to manipulate the variable pool of the currently active REXX procedure.

Shared Variable Request Block

```
/***    Shared Variable Pool Interface */

/***    Function Codes for Variable Pool Interface (shvcode) */

#define RXSHV_SET          0x00       /* Set variable from given value     */
#define RXSHV_FETCH        0x01       /* Copy value of variable to buffer  */
#define RXSHV_DROPV        0x02       /* Drop variable                     */
#define RXSHV_SYSET        0x03       /* Symbolic name Set variable        */
#define RXSHV_SYFET        0x04       /* Symbolic name Fetch variable      */
#define RXSHV_SYDRO        0x05       /* Symbolic name Drop variable       */
#define RXSHV_NEXTV        0x06       /* Fetch "next" variable             */
#define RXSHV_PRIV         0x07       /* Fetch private information          */
#define RXSHV_EXIT         0x08       /* Set function exit value           */
/***    Return Code Flags for Variable Pool Interface (shvret) */

#define RXSHV_OK           0x00     /* Execution was OK                 */
#define RXSHV_NEWV         0x01     /* Variable did not exist           */
#define RXSHV_LVAR         0x02     /* Last var transferred via SHVNEXTV */
#define RXSHV_TRUNC        0x04     /* Truncation occurred during Fetch  */
#define RXSHV_BADN         0x08     /* Invalid variable name            */
#define RXSHV_MEMFL        0x10     /* Out of memory failure            */
#define RXSHV_BADF         0x80     /* Invalid function code (shvcode)   */
/***    Structure of Shared Variable Request Block (SHVBLOCK) */

typedef struct shvnode {
    struct shvnode FAR *shvnext;       /*  Chain pointer to the next block   */
    RXSTRING           shvname;        /*  Pointer to the variable name      */
    RXSTRING           shvvalue;       /*  Pointer to the value buffer       */
    ULONG              shvnamelen;     /*  Length of the name value          */
    ULONG              shvvaluelen;    /*  Length of the fetch value         */
    UCHAR              shvcode;        /*  Individual function code          */
    UCHAR              shvret;         /*  Individual return code flags      */
    } SHVBLOCK;
typedef SHVBLOCK FAR *PSHVBLOCK;

SHORT APIENTRY RxVar (
        PSHVBLOCK);    /* Pointer to a list of SHVBLOCK's             */

#endif /* INCL_RXSHV */
```

Figure 11. Variable Pool Application Programming Interfaces

Interface Function

The calling syntax for the variable pool interface is the following:

RxVar

This call allows the user to pass a linked list of shared variable request blocks to the variable pool.

RxVar (RequestBlockList)

Parameters

RequestBlockList *(PSHVBLOCK) – input*

Is a linked list of shared variable request blocks. Each request block can specify a different operation to be performed by the interpreter. Each request block is processed in turn; control returns to the caller after the last block or after a severe error. In this manner, the interpreter can perform multiple operations by issuing only one call.

An integer return code is returned from the **RxVar** call and contains the return code from the entire set of requests. Return codes for each individual shared variable request block is found in a field of the block.

Returns

0 or positive	Entire shared variable request block list was processed. The return code is the composite OR of the low-order 6 bits of the *shvret* bytes.
-1	Invalid entry conditions.
-2	Insufficient storage was available for a request set. Processing was aborted (some of the request blocks may remain unprocessed; their *shvret* bytes are unchanged.
-3	The variable pool interface was not enabled when this call was issued.
127	The RxVar routine was not found. Equivalent to DosGetProcAddr error_proc_not_found.

Interface Types

Three of the Variable Pool Interface functions (set, fetch, and drop) have dual interfaces.

Symbolic Interface

The symbolic interface refers to normal REXX substitution when interpreting variables. Variable names are valid REXX symbols (in mixed case if desired) including compound symbols. The functions that use the symbolic interface are **RXSHV_SYSET**, **RXSHV_SYFET**, and **RXSHV_SYDRO**.

Direct Interface

The direct interface uses no substitution or case translation. Simple symbols must be valid REXX variable names; in uppercase only and not starting with a digit or a period. Compound symbols, however, allow any characters (including lowercase, blanks, and so on) following a valid REXX stem. The direct interface is used by **RXSHV_SET**, **RXSHV_FETCH**, **RXSHV_DROP**, **RXSHV_NEXTV**, **RXSHV_PRIV**, and **RXSHV_EXIT**.

Shared Variable Functions

The specific actions for each function code are as follows:

Notes:

1. The terms *value length*, *value pointer*, *name length* and *name pointer* will be used generically in the following definitions to describe information derived from the *shvvalue* and *shvname* **RXSTRING** structures.
2. In general terms, the structures *shvname* and *shvvalue* **RXSTRING** are used to describe the memory buffers being passed through the variable pool interface. The *shvnamelen* and *shvvaluelen* lengths are the actual lengths of data processed.

Setting Variables

RXSHV_SET and RXSHV_SYSET—The *name* pointer and length describes the name of the variable to be set, and the *value* pointer and length describes the value that is to be assigned to it. The name is validated to ensure that it does not contain invalid characters, and the variable is then set from the value given. If the name is a stem, all variables with that stem are set, just as though this was a REXX assignment. **RXSHV_NEWV** is set if the variable did not exist before the operation.

Fetching Variables

RXSHV_FETCH and RXSHV_SYFET—The *name* pointer and length describe a buffer containing the name of the variable to be fetched; *shvnamelen* is the length of the name within the buffer. The *value* pointer and length describe a buffer to receive the value of the specified variable. The name is validated to ensure that it does not contain invalid characters, and the variable is then located and copied to the buffer. The total length of the variable is put into *shvvaluelen*. If the value was truncated (because the buffer was not big enough), the **RXSHV_TRUNC** bit is set. If the variable is shorter than the length of the buffer, no padding takes place. If the name is a stem, the initial value of that stem (if any) is returned. **RXSHV_NEWV** is set if the variable did not exist before the operation, and in this case the value copied to the buffer is the derived name of the variable (for example, after substitution). See note 2 on page 190 for additional information.

RXSHV_NEXTV (Fetch next variable)—This function can be used to search through all the variables known to the interpreter (that is, all those of the current generation, excluding those *hidden* by **PROCEDURE** instructions). The order in which the variables are revealed is not specified.

The interpreter maintains a pointer to its list of variables, this is reset to point to the first variable in the list whenever: 1) a host command is issued, or 2) any function other than **RXSHV_NEXTV** is executed by the way of the variable pool interface.

Whenever a **RXSHV_NEXTV** function is executed, the name and value of the next variable available are copied to two buffers supplied by the caller.

The *name* pointer and length describe a buffer in which the name is to be copied. The total length of the name is returned in *shvnamelen*, and if the name was truncated (because the buffer was not big enough), the **RXSHV_TRUNC** bit is set. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the users buffer area using exactly the same protocol as for the Fetch operation.

If *shvret* has **RXSHV_LVAR** set, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers.

If **RXSHV_TRUNC** is set, either the name or the value has been truncated.

By repeatedly running the **RXSHV_NEXTV** function (until the **RXSHV_LVAR** flag is set), a user program may locate all the REXX variables of the current generation. See note 2 on page 190 for additional information.

RXSHV_PRIV (Fetch private information)—This interface is identical to the **RXSHV_FETCH** interface, except that the name refers to certain fixed information items that are available. The entire name must be specified for the fetch to occur. The following names are recognized:

PARM Is the number of parameters (arguments) supplied to the procedure that will be placed in the caller's buffer. The number is formatted as a character string.

PARM.n Is the nth parameter (argument) string that is placed in the caller's buffer. If the nth parameter (argument) is not supplied to the program (whether omitted, null, or fewer than nth parameters were specified), then a null string will be returned.

Note: PARM.1 will return the same result as ARG.

QUENAME
Is the data queue name. The name of the currently active data queue is copied to the user's buffer.

SOURCE Is the source string. The source string, as described on page 48, is copied to the user's buffer.

VERSION
Is the version string. The version string, as described on page 48, is copied to the user's buffer.

See note 2 for additional information.

Dropping Variables

RXSHV_DROPV and RXSHV_SYDRO—The *name* pointer and length describes the name of the variable to be dropped. The *value* pointer and length are not used. The name is validated to ensure that it does not contain invalid characters, and the variable is then dropped, if it exists. If the name given is a stem, all variables starting with that stem are dropped.

Setting an Exit Return Value

RXSHV_EXIT (Set function or exit return value)—An external function written in a language other than REXX or a system exit must have the capability of returning an exit value. A single call is allowed per exit. The value passed on this call is treated like the expression of a RETURN instruction. (See the discussion of the RETURN instruction on page 54.) This interface is identical to the **RXSHV_SET** interface, except that no name is specified. The **RXSHV_EXIT** interface is only valid during processing of an external function or a system exit that allows a string value to be returned.

Notes and Limits

1. The full interface is only enabled during the execution of commands (including CMD subcommands), external routines (functions and subroutines), and some system exits (RXINIT, RXTER, RXCMD, and RXFNC). It is enabled for **RXSHV_EXIT** operations only from exits that allow a character string to be returned (RXHLT, RXMSQ, RXSIO). An attempt to call the RxVar entry point when it is not enabled will result in a return code of -3 (RxVar not enabled).

2. On all fetch-type requests, it is allowable to *not* specify return buffers on the call and have the REXX interpreter allocate and give appropriate memory to the caller. On **RXSHV_FETCH, RXSHV_SYFET**, and **RXSHV_PRIV** the value buffers can be left for REXX to allocate. On **RXSHV_NEXTV**, the value and the name buffers can be allocated by REXX.

   To have REXX supply name buffers, set *shvname* to an empty string for the call. REXX will allocate a memory segment, set up the name pointer and length, and set *shvnamelen* to the length of the name returned. This memory (that is, the far pointer stored in the *shvname* **RXSTRING**) must be freed by the user by way of DosFreeSeg.

   To have REXX supply value buffers, simply set *shvvalue* to an empty string for the call. The REXX interpreter allocates a segments for the value buffer. The *value* pointer and length describes the segments as appropriate for the size of the buffer returned. The total length of the contents of the requested variable is returned in *shvvaluelen*. This memory must be freed by the user by way of DosFreeSeg for the selector stored in the *shvvalue* **RXSTRING**.

   Note that by allowing REXX to supply these buffers, a *RXSHV_TRUNC* condition will not occur. However, it is possible to get *RXSHV_MEMFL* errors for these operations. If *RXSHV_MEMFL* occurs, then no memory will have been allocated for that **SHV_BLOCK** operation.

3. Note that only the main thread of a REXX application is allowed to access the REXX variable pool. Applications can create and use new threads in the course

of performing their function, but only the original thread that started the application can interface with the REXX variable pool.

System Exits

The System Exits interface allows REXX to operate under a user-defined environment. When these exits are used, certain system-dependent interpreter activities can be executed as calls to routines provided by the caller of the interpreter, instead of by the interpreter itself. Except as provided for in the following text, the interpreter does not allow the results of the program being interpreted to be affected by these exits.

There are exits for:

- The administration of resources at the beginning and end of interpretation.
- Assisted linkages to general classes of external facilities (such as, functions)
- Special language features (such as, I/O to standard resources)
- Polling for external interrupts and conditions.

As is the case with subroutine and external-function handlers, system exit routines can be packaged either as dynalink libraries or as EXE modules.

For details on the requirements and options for packaging API handlers, see "General Characteristics" on page 152.

Writing System Exit Handlers

The following is an example of a system exit handler:

```
short pascal far my_sysexit (
     short func,      /* integer code defining the exit function     */
     short subfunc,   /* integer code defining the exit sub-function */
     char far *parm;  /* function dependent control block            */
```

Figure 12. Sample System Exit Handler

Entry Conditions

All exits are passed three parameters on entry. The parameters are as follows:

- The integer code for the exit being invoked
- The subfunction code for the exit
- A far pointer to the exit parameter list.

The parameter list pointed to by the third parameter contains information specific to the exit being invoked. See the following exit descriptions for the format of the parameter list for each exit.

Note: Some of the exits do not have any additional parameters. For those exits, the parameter list pointer will be null.

Exit Conditions

On return from the exit, the integer return value indicates the results of the exit call and the parameter list will have been updated appropriately. The value of the return code can signal one of three actions:

0 Successful handling of the service. The parameter list has been updated as appropriate for that exit.

1 Exit chooses not to handle the service request. The interpreter will handle the request by way of the default means.

-1 A fatal error occurred during processing of this request. REXX error 48, `Failure in system service`, will be raised.

As an example, suppose that you create a routine to process output from the SAY instruction:

- If your exit routine returns 1, SAY processing continues in the normal manner and the output string is displayed on the screen by the REXX interpreter, as usual.
- If your exit routine returns `0`, the interpreter assumes that your exit routine has performed all required output. It therefore bypasses SAY processing, and the output string is not displayed. If your exit routine returns -1, the interpreter halts with error 48, `Failure in system service`.

For many of the exits, an arbitrary length character string can be passed back to the interpreter using the *SHVEXIT* function of the **RxVar** routine. This routine can only be called once per invocation of the exit to return a value.

Identifying Handlers to REXX

System exit handlers must be registered with REXX by way of RxExitRegister before they may be used. This process is only used to give REXX an easy way to handle the routing of exit calls. The registration of System Exit handlers is very similar to the registration of subcommand handlers (the two processes both use the REXX SCBLOCK), but note that only the registration is similar. The calling conventions and handling of system exits is vastly different from subcommands. See the REXXSAA.H file for definitions of the RxExitRegister, RxExitQuery, and RxExitDrop functions.

Enabling System Exits

The interpreter exits are defined on invocation of the interpreter by way of the *sysexit* parameter of the call to the REXX interpreter. The *sysexit* parameter is a pointer to an array of **RXSYSEXIT** structures. Each **RXSYSEXIT** in the array consists of an INT code identifying the exit to be enabled and a pointer to an ASCIIZ string defining the name of a registered system exit environment supplying the system exit routine handler. Create an extra **RXSYSEXIT** element and assign the code to **RXENDLST** to indicate the end of the system exit list.

Interface Functions

The system exit interface (API) functions are similar to a subset of those for the subcommand environment interface. The parameters are the same for the supported functions, and return codes are also comparable. The supported functions are the following:

RxExitRegister

This call allows the user to register a system exit environment.

```
RxExitRegister (SubComBlock)
```

For a description of the parameters, see "RxSubcomRegister" on page 161.

RxExitQuery

This call allows the user to query a system exit environment name.

```
RxExitQuery (EnvName, ModuleName, Flag, UserWord)
```

For a description of the parameters, see "RxSubcomQuery" on page 162.

RxExitDrop

This call allows the user to drop a system exit environment.

```
RxExitDrop (EnvName, ModuleName)
```

For a description of the parameters, see "RxSubcomDrop" on page 165.

Exit Definitions

The following system exits may be specified in the list, with the exit code and subcode definitions shown:

```
#define RXENDLST    0       /* End of exit list.                    */
#define RXFNC    2          /* Process external functions.          */
#define    RXFNCCAL 1       /*    Currently the only subcode value. */
#define RXCMD    3          /* Process host commands.               */
#define    RXCMDHST 1       /*    Currently the only subcode value. */
#define RXMSQ    4          /* Manipulate queue.                    */
#define    RXMSQPLL 1       /*    Pull a line from queue            */
#define    RXMSQPSH 2       /*    Place a line on queue             */
#define    RXMSQSIZ 3       /*    Return number of lines on queue   */
#define    RXMSQNAM 20      /*    Set active queue name (OS/2 only) */

#define RXSIO    5          /* Session I/O.                         */
#define    RXSIOSAY 1       /*    SAY a line to STDOUT              */
#define    RXSIOTRC 2       /*    Trace output                      */
#define    RXSIOTRD 3       /*    Read from session char stream     */
#define    RXSIODTR 4       /*    DEBUG read from session char stream*/
#define    RXSIOTLL 5       /*    Return line length (N/A for OS/2) */

#define RXHLT    7          /* Halt processing.                     */
#define    RXHLTCLR 1       /*    Clear HALT indicator              */
#define    RXHLTTST 2       /*    Test HALT indicator               */
#define RXTRC    8          /* Test external trace indicator.       */
#define    RXTRCTST 1       /*    Currently the only subcode value. */
#define RXINI    9          /* Initialization processing.           */
#define    RXINIEXT 1       /*    Currently the only subcode value. */
#define RXTER   10          /* Termination processing.              */
#define    RXTEREXT 1       /*    Currently the only subcode value. */

#define RXNOOFEXITS 11      /* 1 more than largest exit number.     */

typedef unsigned char far * PEXIT ; /* pointer to exit parameter block*/
```

Figure 13. System Exit Definitions

Each exit has specific characteristics:

- The occasions when the exit is called, if provided.
- The default action to be taken when the exit is not provided.
- The parameter list for the exit. This parameter list is additional to the subcode, which is always passed.
- The action to be taken by the exit.
- The continuation actions to be taken by the interpreter after return from the exit.
- Whether the variable pool interface is enabled during the exit. (It is enabled only for RXCMD, RXFUNC, RXINI, and RXTER; disabled during processing of all other exits).

For many of these exits, an arbitrary length character string can be passed back by way of a **RXSTRING** item in the parameter block. REXX initializes this **RXSTRING** item to an empty string before calling the exit.

If an exit returns a value in its parameter block by way of the *RXSHV_EXIT* function of the **RxVar** routine, REXX uses the value set by the call to **RxVar**. See "Setting an Exit Return Value" on page 190 for details regarding the use of RXVAR and the functions available for the following exits.

RXFNC

Process external functions.

This service supports only one subfunction, RXFNCCAL, whose request code is specified by the second parameter.

RXFNCCAL

Process external functions.

When called: When interpretation is about to call an external routine.

Default action: Call the routine.

Action: Call the routine, if possible.

Continuation: If indicated by the return conditions, raise error 40, 43 or 44. Resume interpretation.

Parameters:

```
typedef struct {
   struct {
      unsigned rxfferr  : 1;          /* Invalid call to routine.    */
      unsigned rxffnfnd : 1;          /* Function not found.         */
      unsigned rxffsub  : 1;          /* Called as a subroutine if   */
                                      /* set.  Return values are     */
                                      /* optional for subroutines,   */
                                      /* required for functions.     */
   } rxfnc_flags ;

   PUCHAR           rxfnc_name;       /* Pointer to function name.   */
   USHORT           rxfnc_namel;      /* Length of function name.    */
   PUCHAR           rxfnc_que;        /* Current queue name.         */
   USHORT           rxfnc_quel;       /* Length of queue name.       */
   USHORT           rxfnc_argc;       /* Number of args in list.     */
   PRXSTRING        rxfnc_argv;       /* Pointer to argument list.   */
                                      /* List mimics argv list in    */
                                      /* REXXSAA -- array of         */
                                      /* RXSTRINGs.                  */
   RXSTRING         rxfnc_retc;       /* Return value.               */
} RXFNCCAL_PARM;
```

Upon entry to RXFNCCAL, the name of the function to be invoked is defined by *rxfnc_name* and *rxfnc_namel*. The arguments to the function are indicated by *rxfnc_argc* and *rxfnc_argv*. If the named routine is invoked by a CALL instruction (rather than as a function call), then the flag *rxffsub* is set *on*.

On return from the exit, values in *rxfnc_flags* indicate the status of the processing of the function. If neither *rxfferr* or *rxffnfnd* is on, then the routine has been successfully invoked and has run successfully. The error flags are checked only if the exit returns 0.

The *rxffnfnd* flag indicates that the exit could not locate the routine with the given name. The interpreter will give error 43, `Routine not found`. The *rxfferr* flag

indicates that the parameters supplied to the routine were somehow invalid. The interpreter will give error 40, `Invalid call to routine.`.

Note: The variable pool interface is fully enabled during calls to the RXFNC exits. Use the *RXSHV_EXIT* function to return the value of the function named in *rxfnc_name*, or return the value by way of the *rxfnc_retc* field of the exit parameter list. This routine can be called only once per invocation of the exit to return a value. If the routine was invoked as a function and a result was not returned, then the interpreter will give error 44, `Function did not return data`. If the routine was invoked as a subroutine, then the returned result is optional.

RXCMD

Process host commands.

This service supports only one subfunction, RXCMDHST, whose request code is specified by the second parameter.

RXCMDHST

Process host commands.

When called: When interpretation is about to pass a command string to some external environment.

Default action: Pass the command to the external environment.

Action: Pass the command to the external environment, if possible.

Continuation: If indicated by the results, raise the ERROR or FAILURE condition. Resume interpretation.

Parameters:

```
typedef struct {
   struct {                              /* Condition flags            */
      unsigned rxfcfail : 1;             /* Command failed.  Trap with  */
                                         /* CALL or SIGNAL on FAILURE.  */
      unsigned rxfcerr  : 1;             /* Command ERROR occurred.     */
                                         /* Trap with CALL or SIGNAL on */
                                         /* ERROR.                      */
   } rxcmd_flags;
   PUCHAR           rxcmd_address;       /* Pointer to address name.    */
   USHORT           rxcmd_addressl;      /* Length of address name.     */
   PUCHAR           rxcmd_dll;           /* dll name for command.       */
   USHORT           rxcmd_dll_len;       /* Length of dll name.  0 ==>  */
                                         /* .EXE file.                  */
   RXSTRING         rxcmd_command;       /* The command string.         */
   RXSTRING         rxcmd_retc;          /* Pointer to return code      */
                                         /* buffer.  User allocated.    */
} RXCMDHST_PARM;
```

On entry to the exit, *rxcmd_retc* defines a buffer that is used to return a value to use for the return code in character format (that is, numeric return codes must be formatted as a character string). The return code may be a non-numeric value if desired. The *rxcmd_command* parameter contains the command string itself. The *rxcmd_dll* and *rxcmd_dll_len* parameters define the name of the dynalink library specified in the ADDRESS instruction.

The *rxfcfail* and *rxfcerr* flags are used by this exit to indicate that an ERROR or FAILURE condition has occurred. The definition of what constitutes an ERROR or FAILURE condition of a command is under the control of the exit.

Note: The variable pool interface and all subcommand-registration functions are fully enabled during calls to the RXCMD exits.

RXMSQ

Manipulate queue.

This service supports a number of subfunctions. The subfunction request code is specified by the second parameter. The parameter list depends on the particular subfunction invoked. The RXMSQ subfunctions and their parameter lists are:

RXMSQPLL

Pull a line from the queue.

When called: Used to interpret PULL.

Default action: Pull from the current default queue.

Action: Provide a result.

Continuation: Resume interpretation, using the result.

Parameters:

```
typedef struct {
   RXSTRING          rxmsq_retc;       /* Pointer to dequeued entry   */
                                       /* buffer.  User allocated.    */
} RXMSQPLL_PARM;
```

This exit is invoked by the PULL and PARSE PULL instructions. Upon entry to RXMSQPLL, *rxmsq_retc* defines a buffer that is used to return a value to use for the line removed from the queue.

RXMSQPSH

Place a line on the queue.

When called: Used to interpret PUSH and QUEUE.

Default action: Push the value on the current default queue.

Action: Push the value on some queue where RXMSQPLL can subsequently fetch it.

Continuation: Resume interpretation.

Parameters:

```
typedef struct {
   struct {                            /* Operation flag              */
      unsigned rxfmlifo : 1;           /* Stack entry LIFO if set,    */
                                       /* FIFO if reset.              */
   } rxmsq_flags;

   RXSTRING          rxmsq_value;      /* The entry to be pushed.     */
} RXMSQPSH_PARM;
```

The line to be placed on the queue (*rxmsq_value*) is the result of evaluating the expression specified on a PUSH or QUEUE instruction. It is the responsibility of the exit to handle truncation of this string if the exit has a restriction on the maximum width of the queue. The stacking order is indicated by the *rxfmlifo* flag.

RXMSQSIZ

Return the number of lines in the queue.

When called: Used to interpret the QUEUED built-in function.

Default action: Find the size of the current default queue.

Action: Return the size of the queue from which RXMSQPLL pulls.

Continuation: Use the returned value as the value of QUEUED. Resume interpretation.

Parameters:

```
typedef struct {
   ULONG            rxmsq_size;       /* Number of Lines in Queue    */
} RXMSQSIZ_PARM;
```

This exit is invoked by the QUEUED function and by the PULL and PARSE PULL instructions.

On return, *rxmsq_size* contains the the number of lines in the data queue as an integer.

RXMSQNAM

Set the name of the active queue.

When called: Used to interpret **RXQUEUE ("SET",** *newname*) operating system specific routine.

Default action: Change the current default queue to *newname*.

Action: Change the queue to be operated upon by RXMSQPLL, RXMSQPSH, and RXMSQSIZ.

Continuation: Resume interpretation.

Parameters:

```
typedef struct {
   PSZ              rxmsq_name;       /* Selector containing ASCIIZ  */
                                      /* queue name.  Change length  */
                                      /* with DosReallocSeg if       */
                                      /* required.                   */
} RXMSQNAM_PARM;
```

This exit is invoked by the **RXQUEUE ("SET",** *newqueuename*) function. See "RXQUEUE Function" on page 145.

RXSIO

Session I/O.

This service supports the following subfunctions. The subfunction request code is specified by the second parameter. The parameter list depends on the particular subfunction invoked. The following are the RXSIO subfunctions and their parameter lists.

RXSIOSAY

To output the result of a SAY instruction to the standard output stream.

When called: To output the result of a SAY instruction.

Default action: Write to the session standard output device.

Action: Send the line.

Continuation: Resume interpretation.

Parameters:

```
typedef struct {
   RXSTRING          rxsio_string;     /* String to display.          */
} RXSIOSAY_PARM;
```

The line to be delivered to the standard output device is the result of evaluating the expression specified on a SAY instruction. This string can be any length up to the size of standard output device (as determined by default system processing). It is the responsibility of the exit to handle truncation of this string if the string is too long.

RXSIOTRC

TRACE output processing. Call to output TRACE results.

When called: To output the result of each line of tracing.

Default action: Write to the session standard error device.

Action: Send the line.

Continuation: Resume interpretation.

Parameters:

```
typedef struct {
  RXSTRING           rxsio_string;     /* Trace line to display.      */
} RXSIOTRC_PARM;
```

The line to be displayed at the standard error-output device is the result of a traced line. The form of the string contained in *rxsio_string* is:

1. Characters 1 though 6
2. The line number (with leading spaces)
3. A space
4. Followed by the traced program line
5. Ended by a semicolon.

RXSIOTRD

Read from STDIN stream.

When called: To read the standard input stream (STDIN). Note that if PULL was satisfied by something from the stack then this routine is not called.

Default action: Read from the default character input stream.

Action: Read a line.

Continuation: Resume interpretation

Parameters:

```
typedef struct {
   RXSTRING          rxsiotrd_retc;    /* RXSTRING for output.  Note: */
                                       /* user allocates output       */
                                       /* buffer with DosAllocSeg()   */
                                       /* or DosAllocHuge().          */
} RXSIOTRD_PARM;
```

This exit is invoked by the instructions PULL and PARSE PULL when the active data queue is empty. (The PARSE LINEIN instruction does not invoke this exit.)

Upon entry, *rxsio_retc* defines a **RXSTRING** item that is used to return a value to use for the line read from STDIN. REXX initializes this field to be a **RXSTRING** item having a 250-character buffer. If additional length is needed, the system exit must allocate the output buffer with DosAllocSeg or DosAllocHuge.

RXSIODTR

Debug read.

When called: To read from the session character stream when for interactive debug. This routine differs from RXSIOTRD since it is not dependent on the state of the queue.

Default action: Read from the default character input stream.

Action: Read a line.

Continuation: Resume interpretation

Parameters: Debug read from STDIN: stream.

```
typedef struct {
   RXSTRING          rxsiotrd_retc;    /* RXSTRING for output.  Note: */
                                       /* user allocates output       */
                                       /* buffer with DosAllocSeg()   */
                                       /* or DosAllocHuge().          */
} RXSIODTR_PARM;
```

This exit is invoked by STDIN input during interactive debugging (invoked by the TRACE? instruction; see "Interactive Debugging of Programs" on page 205.)

Upon entry, *rxsio_retc* defines a **RXSTRING** item that is used to return a value to use for the line read from the STDIN stream. REXX initializes this field to be a **RXSTRING** item having a 250-character buffer. If additional length is needed, the system exit must allocate the output buffer with DosAllocSeg or DosAllocHuge.

RXHLT

Halt processing.

This service supports two subfunctions. The request code for each subfunction is specified by the second parameter. The parameter list depends on the particular subfunction invoked.

RXHLTCLR

Clear Halt indicator.

This exit subfunction has no inputs or outputs. It is invoked by the interpreter to reset the external flag after RXHLTTST has returned a flag value of 1.

When called: After raising HALT and before the next (potential) call of RXHLTTST.

Default action: The interpreter makes use of the system facilities for resetting the polling of external interrupts.

Action: Reset the polling of the external interrupt.

Continuation: Continue interpretation.

Parameters: None.

RXHLTTST

Test Halt indicator.

When called: RXHLTTST is called by the interpreter to poll whether an external attempt has been made to interrupt the execution. The interpreter must make sufficient calls to ensure that any execution that could be halted by this mechanism is eventually halted.

Default action: The interpreter makes use of the system facilities for polling external interrupts.

Action: Set the result to indicate whether the external interrupt has occurred.

Continuation: Raise the HALT condition if the interrupt has occurred. Continue interpretation.

Parameters:

```
typedef struct {
   struct {                          /* Halt flag                  */
      unsigned rxfhhalt : 1;         /* Set if HALT occurred.      */
   } rxhlt_flags;
} RXHLTTST_PARM;
```

This exit informs the REXX interpreter when a HALT condition has been raised. It is invoked at the end of each clause. Upon return, the *rxfhhalt* flag will indicate whether a HALT condition has occurred (1 = Halt, 0 = No Halt). RXHLTTST may also return a string that is available by way of the built-in function CONDITION(D). This string is returned by way of the use of the *SHVEXIT* function of the RXVAR routine. This routine can be called only once per invocation of the exit to return a value.

RXTRC

Test external trace indicator.

This service supports only one subfunction, whose request code is specified by the second parameter.

RXTRCTST

Test external trace indicator.

When called: The tracing feature of REXX can be controlled externally to the program. RXTRC is called by the interpreter to poll whether the external indication says that trace is active.

Default action: The interpreter makes use of the system facilities for indicating this condition.

Action: Set the result to indicate whether the external condition is set.

Continuation: Continue with or without tracing, according to the result.

Parameters:

```
struct rxtrc_parm {
  struct {
    unsigned rxftrace : 1;        /* External trace setting        */
         } rxtrc_flags;
}
```

This exit is invoked at the end of each clause. Upon return from this exit, *rxftrace* will indicate whether an external trace condition has occurred (1 = External Trace, 0 = No External Trace).

Note: If the *rxftrace* flag is set from 0 to 1, then on return, REXX will begin interactive debug mode. If the *rxftrace* flag is set from 1 to 0, then trace will turn off interactive debug mode.

RXINI

Initialization processing.

This service supports only one subfunction, whose request code is specified by the second parameter.

RXINIEXT

Initialization processing.

This exit has no inputs or outputs. It is called before the first instruction of the program is interpreted.

Note: The variable pool interface is enabled when this exit is called.

When called: Before the last instruction of the subject program is interpreted.

Default action: None.

Action: This is an opportunity for the system to adapt to the change of context; that is, to initialize so that the other user exits will work.

Continuation: Resume interpretation.

Parameters: None.

RXTER

Termination processing.

This service supports only one subfunction, whose request code is specified by the second parameter.

RXTEREXT

Termination processing.

This exit has no inputs or outputs. It is called after the last instruction of the program is interpreted. If the termination exit fails (that is, returns -1), no action is taken.

Note: The variable pool interface is enabled when this exit is called.

When called: After the last instruction of the subject program is interpreted.

Default action: Do nothing

Action: This is an opportunity for the system to adapt to the change of context.

Continuation: Resume, leaving the interpreter.

Parameters: None.

Chapter 10. Debugging Aids

In addition to the TRACE instruction described on page 60, there are the following debugging aids.

Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a program. Adding the prefix character ? to the TRACE instruction or the TRACE function (for example, `TRACE ?I` or `TRACE(?I)`) turns on interactive debug and indicates to the user that interactive debug is active. Further TRACE instructions in the program are ignored, and the language processor pauses after nearly all instructions that are traced at the console (see the following for the exceptions). When the language processor pauses, three debug actions are available:

1. **Entering a null line** makes the language processor continue execution until the next pause for debug input. Repeatedly entering a null line, therefore, steps from pause point to pause point. For `TRACE ?A`, for example, this is equivalent to single-stepping through the program.
2. **Entering an equal sign (=)** with no blanks makes the language processor re-execute the clause last traced. For example, if an IF clause is about to take the wrong branch, you can change the value of the variables on which it depends, and then re-execute it.

   Once the clause has been re-executed, the language processor pauses again.
3. **Anything else entered** is treated as a *line* of one or more clauses, and processed immediately (that is, as though DO; *line*; END; had been inserted in the program). The same rules apply as in the INTERPRET instruction (for example, DO-END constructs must be complete). If an instruction has a syntax error in it, a standard message is displayed and you are prompted for input again. Similarly all the other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that nonzero return codes from host commands are displayed. Host commands are always executed, but the variable RC is not set. Once the string has been processed, the language processor pauses again for further debug input.

Interactive debug is turned off:

- If a TRACE instruction uses the ? prefix while interactive debug is in effect

  or

- At any time, if `TRACE 0` or `TRACE` with no options is entered.

The numeric form of the TRACE instruction may be used to allow sections of the program to be executed without pause for debug input. `TRACE n` (that is, positive result) allows execution to continue, skipping the next n pauses (when interactive debug is or becomes active). `TRACE -n` (that is, negative result) allows execution to continue without pause and with tracing inhibited for n clauses that would otherwise be traced. The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through a program (say after using `TRACE ?R` to trace Results) and then enter a subroutine in which you have

© Copyright IBM Corp. 1983, 1990

no interest, you can enter TRACE 0 to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Similarly, if you are interested only in a subroutine, you can put a TRACE ?R instruction at its start. Having traced the routine, the original status of tracing is restored and hence (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

Since any instructions may be executed in interactive debug you have considerable control over execution.

The following are some examples:

```
Say expr      /* displays the result of evaluating the      */
              /* expression.                                */

name=expr     /* alters the value of a variable.            */

Trace 0       /* (or Trace with no options) turns off       */
              /* interactive debug and all tracing.         */

Trace ?A      /* turns off interactive debug but continue   */
              /* tracing all clauses.                       */

exit          /* terminates execution of the program.       */

Do i=1 to 10 /* displays ten elements of the array stem.    */
say stem.i
end
```

Exceptions: Some clauses cannot safely be re-executed, and therefore the language processor does not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop.
- All END clauses (not a useful place to pause in any case).
- All THEN, ELSE, OTHERWISE, or null clauses.
- All RETURN clauses, except when returning from an internal function or subroutine call.
- All EXIT clauses.
- All SIGNAL and CALL clauses (the language processor pauses after the target label has been traced).
- Any clause that causes a syntax error. (These may be trapped by SIGNAL ON SYNTAX, but cannot be re-executed.)

RXTRACE Variable

When the interpreter starts the interpretation of a REXX procedure it will check the setting of the special environment variable, *RXTRACE*. If *RXTRACE* has been set to ON (not case sensitive) then the interpreter will start in interactive debug mode (as if the REXX instruction TRACE '?R' had been the first interpretable instruction). All other settings of *RXTRACE* will be ignored. *RXTRACE* will only be checked when starting a new REXX procedure.

Chapter 11. Reserved Keywords and Special Variables

Keywords may be used as ordinary symbols in many situations where there is no ambiguity. The precise rules are given in this chapter.

There are three special variables: RC, RESULT, and SIGL.

Reserved Keywords

The free syntax of REXX implies that some symbols are reserved for use by the language processor in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of REXX keywords are the WHILE keyword in a DO instruction, and the THEN keyword, which acts as a clause terminator in this case, following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords; the symbols may be freely used elsewhere in clauses without being understood as keywords.

Be careful of host commands or subcommands with the same name as REXX keywords (for example, the OS/2 command CALL). This can create problems for any programmer whose REXX programs might be used for some time and in circumstances outside his or her control, and who wishes to make the program absolutely *watertight.*

In this case, a REXX program may be written with at least the first words in command lines enclosed in quotes.

The following is an example:

```
'DELETE' Fn'.'Ext
```

This also has an advantage in that it is more efficient; and with this style, the SIGNAL ON NOVALUE condition may be used to check the integrity of an exec.

An alternative strategy is to precede such command strings with two adjacent quotes, that has have the effect of concatenating the null string on to the front.

The following is an example:

```
''Erase Fn'.'Ext
```

A third option is to enclose the entire expression, or the first symbol, in parentheses.

The following is an example:

```
(Erase Fn'.'Ext)
```

The choice of strategy, if it is to be done at all, is a personal one by the programmer. It is not imposed by the REXX language.

© Copyright IBM Corp. 1983, 1990

Special Variables

There are three special variables that may be set automatically by the language processor:

RC Is set to the return code from any executed host command (or subcommand). Following the SYNTAX, ERROR, and FAILURE SIGNAL events, RC is set to the code appropriate to the event; the syntax error number (see Appendix A for error messages) or the command return code. RC is unchanged following a NOVALUE or HALT event.

Note: Host commands that are executed manually from debug mode do not cause the value of RC to change.

RESULT Is set by a RETURN instruction in a subroutine that has been called if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it, RESULT is dropped (becomes uninitialized.)

SIGL Contains the line number of the clause currently running when the last transfer of control to a label took place. (This could be caused by a SIGNAL event, a CALL instruction, an internal function invocation, or a trapped error condition.)

None of these variables has an initial value. They may be altered by the user, just like any other variable. They also may be accessed by way of the variable pool interface (see "Variable Pool Interface" on page 186). The PROCEDURE and DROP instructions also affect these variables in the usual way.

Certain other information is always available to a REXX program. This includes the name by which the program was invoked and the source of the program (which is available using the PARSE SOURCE instruction, see page 48). The latter consists of the string "OS/2," followed by the call type and then the full path specification of the file being executed.

In addition, PARSE VERSION (see page 48) makes available the version and date of the language processor code that is running. The built-in functions TRACE and ADDRESS return the current trace setting and environment name respectively.

Finally, the current settings of the NUMERIC function can be obtained using the DIGITS, FORM, and FUZZ built-in functions.

Chapter 12. Useful OS/2 Commands

CALL Command

The OS/2 CALL command should not be confused with the REXX CALL *instruction* (see page 28), which is used exclusively within REXX programs to invoke internal or external routines.

- **When used in an OS/2 batch file,** the CALL command invokes a REXX program or another batch file. This allows REXX programs (and other batch files) to be used as commands from a master batch file. Upon completion of the called routine, execution returns to the calling batch file.

  Invoking a REXX program or batch file by name only (without CALL) from within a batch file *transfers* control to the named routine, without returning execution to the calling batch file.

- **In a REXX program,** the OS/2 CALL command can be used, in a REXX command expression, to call another REXX program as a *subcommand* (rather than as a subroutine or function), or to call a batch file. This causes a new invocation of CMD.EXE (and, if a REXX program the REXX interpreter). Upon completion of the called program, control returns to the calling program (and the original invocation of of the interpreter). For example, to call a REXX program `prog1` as a subroutine, use the CALL instruction

  ```
  call prog1
  ```

  To call `prog1` as an OS/2 subcommand, put the calling expression in quotes to signify a command expression to be passed to the OS/2 program:

  ```
  "call prog1"
  ```

Notes:

1. Using the REXX program name alone as command expression (for example, `"prog1"` without CALL) can cause a chaining error to occur.
2. The OS/2 CALL command does not allow a CMD file to call itself. If it does, it runs out of stack space and ends.

Other Commands

COPY	Copies a file.
DELETE	Deletes a file.
DIR	Displays listing of files in the current directory.
MODE	Controls various characteristics of input and output devices.
PATH	Defines the directory paths to search for commands or REXX programs, if not found in the current or a specified directory. See also "Search Order" on page 66.
SET	Displays or changes the OS/2 environment settings. See also "VALUE" on page 104.

© Copyright IBM Corp. 1983, 1990

Applications Programming Interfaces

Subcommand Environment Services

For a full description of subcommand environment services, see "Subcommand Interface" on page 157.

RXSUBCOM REGISTER *envname dllname procname*
Registers a subcommand environment.

RXSUBCOM DROP *envname* [*dllname*]
Deregisters a subcommand environment.

RXSUBCOM LOAD *envname* [*dllname*]
Loads a subcommand environment.

RXSUBCOM QUERY *envname* [*dllname*]
Queries a subcommand environment name.

Queue Services (Filters)

The RXQUEUE filter normally operates on the default queue named SESSION. However, if an environment variable named "RXQUEUE" exists, the value of that variable is used to determine the name of the queue to use.

For a full description of REXX queue services for applications programming, see "Queue Interface" on page 144.

RXQUEUE [*queuname*] **/LIFO**
Stacks items from STDIN on the top of a named (or default) REXX queue.

RXQUEUE [*queuname*] **/FIFO**
Queues items from STDIN to the end of a named (or default) REXX queue.

RXQUEUE [*queuname*] **/CLEAR**
Clears the named (or default) REXX queue.

Appendix A. Error Numbers and Messages

The error numbers produced by syntax errors during processing of REXX programs are all in the range 3 to 49 (and this is the value placed in the variable RC when SIGNAL ON SYNTAX event is trapped).

Three of the error messages can be generated by the external interfaces to the language processor either before the language processor gains control or after control has left the language processor. Therefore these errors cannot be trapped by SIGNAL ON SYNTAX. The error numbers involved are 3 and 5 (if the initial requirements for storage could not be met) and 26 (if on exit, the returned string could not be converted to form a valid return code). Similarly, error 4 can be trapped only by SIGNAL ON HALT or CALL ON HALT.

Error 1 **File table full**

Explanation: There are currently too many files open for this session. Close any files that are open but that are not in use.

Error 2 **Interpret expression > 64000 characters**

Explanation: An attempt has been made to interpret a clause that exceeded the 64000 character limit. Split the clause into two or more equivalent clauses.

Error 3 **Program is unreadable**

Explanation: The REXX program could not be read from the disk.

Error 4 **Program interrupted**

Explanation: The system interrupted execution of your REXX program.

Error 5 **Machine resources exhausted**

Explanation: While attempting to process a program, the language processor was unable to get the space needed for its work areas and variables. This may have occurred because the program that invoked the language processor has already used up most of the available storage itself.

Error 6 **Unmatched "/*" or quote**

Explanation: The language processor reached the end of the file (or the end of data in an INTERPRET instruction) without finding the ending "*/" for a comment or quote for a literal string.

Error 7 **WHEN or OTHERWISE expected**

Explanation: The language processor expected a series of WHEN expressions and an OTHERWISE within a SELECT statement. This message is issued when any other instruction is found or if all WHEN expressions are found to be false and an OTHERWISE is not present. This error is often caused by forgetting the DO and END instructions around the list of instructions following a WHEN. For example:

```
  WRONG                    RIGHT

Select                   Select
  When a=b then            When a=b then DO
    Say 'A equals B'         Say 'A equals B'
    exit                     exit
  Otherwise nop              end
  end                      Otherwise nop
                           end
```

Error 8 **Unexpected THEN or ELSE**

Explanation: The language processor has found a THEN or an ELSE clause that does not match a corresponding IF clause. This situation is often caused by forgetting to put an END or DO END instruction in the THEN part of a complex IF THEN ELSE construction. For example:

```
  WRONG                    RIGHT

If a=b then do;           If a=b then do;
  Say EQUALS                Say EQUALS
  exit                      exit
 else                       end
  Say NOT EQUALS           else
                             Say NOT EQUALS
```

© Copyright IBM Corp. 1983, 1990

Error 9 Unexpected WHEN or OTHERWISE

Explanation: The language processor has found a WHEN or OTHERWISE instruction outside of a SELECT construction. You may have accidentally enclosed the instruction in a DO END construction by leaving off an END instruction, or you may have tried to branch to it with a SIGNAL statement (which cannot work because the SELECT construction is then terminated).

Error 10 Unexpected or unmatched END

Explanation: The language processor has found more END instructions in your program than DO or SELECT instructions, or the END instructions were placed so that they did not match the DO or SELECT instructions.

This message can be caused if you try to SIGNAL into the middle of a loop. In this case, the END instruction will be unexpected because the previous DO instruction will not have been executed. Remember also, that SIGNAL terminates any current loops, so it can not be used to jump from one place inside a loop to another.

This message can also be caused if you place an END instruction immediately after a THEN or ELSE construction. It may be helpful to use TRACE Scan to show the structure of the program and make it more obvious where the error is. Putting the name of the control variable on END instructions that close repetitive loops can also help locate this kind of error.

Error 11 Control stack full

Explanation: This message is issued if you exceed the limit on levels of nesting of control structures (DO-END, IF-THEN-ELSE, and so on).

This message could be caused by a looping INTERPRET instruction. For example:

```
line='INTERPRET line'
INTERPRET line
```

These lines would loop until they exceeded the nesting level limit and this message would be issued. Similarly, a recursive subroutine that does not terminate correctly could loop until it causes this message.

Error 12 Clause too long

Explanation: You have exceeded the character-length limit for the internal or external representation of a clause.

If the cause of this message is not obvious to you, it may be due to a missing quote that has caused a number of lines to be included in one long string. In this case, the error probably occurred at the start of the data included in the clause traceback (flagged by + + + in the output).

The internal representation of a clause does not include comments or multiple blanks that are outside of strings.

Error 13 Invalid character in program

Explanation: The language processor found an invalid character outside of a literal (quoted) string. Valid characters are:

```
A through Z a through z 0 through 9
(Alphamerics)

? .  (Name Characters)

& * ( ) - + = ¬ ' " ; : < , > / \ ¦
(Special Characters)
```

Error 14 Incomplete DO/SELECT/IF

Explanation: The language processor has reached the end of the file (or end of data for an INTERPRET instruction) and has found that there is a DO or SELECT instruction without a matching END instruction, or an IF clause that is not followed by a THEN clause. Putting the name of the control variable on END instructions that close repetitive loops can help locate this kind of error.

Error 15 Invalid hexadecimal or binary string

Explanation: For the language processor, hexadecimal or binary constants cannot have leading or trailing blanks and can have imbedded blanks only at byte boundaries. The following are all valid hexadecimal constants:

```
'13'x
'A3C2 1c34'x
'1de8'x
```

These are all valid binary constants:

```
'1011'b
'110 1101'b
'101101 11010011'b
```

You may have mistyped one of the digits, for example, typing a letter o instead of a 0. This message can also be caused if you follow a string by the 1-character symbol X or B (as the name of a variable *X* or *B*) when the string is not intended to be taken as a hexadecimal or binary specification. In this case, use the explicit concatenation operator (||) to concatenate the string to the value of the symbol.

Error 16 Label not found

Explanation: The language processor could not find the label specified by a SIGNAL instruction or a label matching an enabled condition when the corresponding (trapped) event occurred. You may have mistyped the label or forgotten to include it.

Error 17 Unexpected PROCEDURE

Explanation: The language processor encountered a PROCEDURE instruction in an invalid position. This could occur because no internal routines are active or because the PROCEDURE instruction was not the first instruction executed after the CALL instruction or function invocation. This error can be caused by *dropping through* to an internal routine, rather than invoking it with a CALL instruction or a function call.

Error 18 THEN expected

Explanation: Each REXX IF and WHEN clause must be followed by a THEN clause. Another clause was found before a THEN clause was found.

Error 19 String or symbol expected

Explanation: The language processor expected a symbol or string following the CALL or SIGNAL keywords, but none was found. You may have omitted the string or symbol, or you may have inserted a special character (such as a parenthesis).

Error 20 Symbol expected

Explanation: The language processor expected a symbol following the CALL ON, END, ITERATE, LEAVE, NUMERIC, PARSE, PROCEDURE, or SIGNAL ON keywords or expected a list of symbols following the DROP or PROCEDURE (with EXPOSE option) keywords. Either there was no symbol when one was required or some other characters were found.

Error 21 Invalid data on end of clause

Explanation: You have followed a clause, such as SELECT or NOP, by some data other than a comment.

Error 23 Invalid character string.

Explanation: A data string (that is, the result of an expression) contains character codes that are not valid in the interpreter. This might be because some characters are *impossible* or because the character set is extended in some way and a given character combination is not allowed.

Error 24 Invalid TRACE request

Explanation: The language processor issues this message when the action specified on a TRACE instruction or the argument to the built-in function, did not start with an A, C, E, F, I, L, N, O, or R.

Error 25 Invalid subkeyword found

Explanation: The language processor expected a particular subkeyword at this position in an instruction but something else was found. For example, the NUMERIC instruction must be followed by the DIGITS, FUZZ, or FORM subkeyword. If NUMERIC is followed by anything else, this message will be issued.

Error 26 Invalid whole number

Explanation: The language processor found an expression in the NUMERIC instruction, a parsing positional pattern, or the right hand term of the exponentiation (**) operator that did not evaluate to a whole number or was greater than the limit, for these uses, of 999 999 999. This error condition is also raised when the DO repetitor is not a positive whole number or when an integer-divide or remainder operation does not result in a whole number.

Error 27 Invalid DO syntax

Explanation: The language processor found a syntax error in the DO instruction. You might have used BY, TO, or FOR phrases twice, or used BY, TO, or FOR instruction when you did not specify a control variable.

Error 28 Invalid LEAVE or ITERATE

Explanation: The language processor encountered an invalid LEAVE or ITERATE instruction. The instruction was invalid because:

- No loop was active

  or

- The name specified on the instruction did not match the control variable of any active loop.

Note that internal routine calls and the INTERPRET instruction protect DO loops by making them inactive. Therefore, for example, a LEAVE instruction in a subroutine cannot affect a DO loop in the calling routine.

You can cause this message to be issued if you use the SIGNAL instruction to transfer control within or into a loop. A SIGNAL instruction terminates all active loops, and any ITERATE

or LEAVE instruction issued then would cause this message to be issued.

Error 29 Environment name too long

Explanation: The language processor encountered an environment name specified on an ADDRESS instruction that is longer than the allowed limit of 250 characters.

Error 30 Name or string too long

Explanation: The language processor found a variable name or a literal (quoted) string that is longer the allowed limit of 250 characters.

The limit for names includes any substitutions. A possible cause of this error is the use of a period (.) in a name, causing an unexpected substitution.

For a literal string, this error can be caused by leaving off an ending quote (or putting a single quote in a string) because several clauses may be included in the string. For example, the string `'don't'` should be written as `'don''t'` or `"don't"`.

Error 31 Name starts with numeric or "."

Explanation: The language processor found a variable whose name began with a digit or a period. The REXX rules do not allow you to assign a value to a variable whose name begins with a digit or a period, because you could then redefine numeric constants.

Error 33 Invalid expression result

Explanation: The language processor encountered an expression result that is invalid in its particular context. The result may be invalid because an illegal FUZZ or DIGITS value was used in a NUMERIC instruction (FUZZ must be no larger than DIGITS). Another possibility is an invalid or missing expression following a VALUE keyword in an instruction.

Error 34 Logical value not 0 or 1

Explanation: The language processor found an expression in an IF, WHEN, DO WHILE, or DO UNTIL phrase that did not result in a `0` or 1. Any value operated on by a logical operator (¬, ¦, &, or &&) must result in a `0` or 1. For example, the phrase `If result then exit rc` will fail if `result` has a value other than `0` or 1. Thus, the phrase would be better written as `If result¬=0 then exit rc`.

Error 35 Invalid expression

Explanation: The language processor found a grammatical error in an expression. You might have ended an expression with an operator, had two adjacent operators with no data in between, or included special characters (such as operators) in an intended character expression without enclosing them in quotes. For example, in the OS/2 program, the command DIR C:\UTIL*.* should be written as `DIR 'C:\UTIL\*.*'` (assuming DIR is not a variable) or even as `'DIR C:\UTIL\*.*'`.

Error 36 Unmatched "(" in expression

Explanation: The language processor found an unmatched parenthesis within an expression. You will get this message if you include a single parenthesis in a command without enclosing it in quotes.

Error 37 Unexpected " ," or ")"

Explanation: The language processor found a comma (,) outside a routine invocation or too many right parentheses in an expression. You will get this message if you include a comma in a character expression without enclosing it in quotes. For example, the instruction:

```
Say Enter A, B, or C
```

should be written as:

```
Say 'Enter A, B, or C'
```

Error 38 Invalid template or pattern

Explanation: The language processor found an invalid special character, for example, % within a parsing template, or the syntax of a variable trigger was incorrect (no symbol was found after a left parenthesis). This message is also issued if the WITH subkeyword is omitted in a PARSE VALUE instruction.

Error 39 Evaluation stack overflow

Explanation: The language processor was not able to evaluate the expression because it is too complex (many nested parentheses, functions, and so on).

Error 40 Incorrect call to routine

Explanation: The language processor encountered an incorrect call to a built-in or external routine. Some possible causes are:

- You passed invalid data (arguments) to the routine.
- You passed too many arguments to the routine.
- The external routine invoked was not compatible with the language processor.

If you were not trying to invoke a routine, you may have a symbol or a string adjacent to a "(" when you meant it to be separated by a space or an operator. This causes it to be seen as a function call. For example, `TIME(4+5)` should probably be written as `TIME*(4+5)`.

Error 41 Bad arithmetic conversion

Explanation: The language processor found a term in an arithmetic expression that was not a valid number or that had an exponent outside the allowed range of -999 999 999 to +999 999 999.

You may have mistyped a variable name, or included an arithmetic operator in a character expression without putting it in quotes. For example, the command `DIR *prod.dat` should be written as `'DIR *prod.dat'` (in quotes); otherwise, the language processor will try to multiply `DIR` by `prod.dat`.

Error 42 Arithmetic overflow/underflow

Explanation: The language processor encountered the result of an arithmetic operation that required an exponent greater than the limit of nine digits (more than 999 999 999 or less than -999 999 999).

This error can occur during evaluation of an expression (often as a result of trying to divide a number by 0) or during the stepping of a DO loop control variable.

Error 43 Routine not found

Explanation: The language processor was unable to find a routine called in your program. You invoked a function within an expression or invoked a subroutine by CALL, but:

- The specified label is not in the program.
- It is not the name of a built-in function.
- The language processor could not locate it externally.

The simplest, and probably most common, cause of this error is mistyping the name.

If you were not trying to invoke a routine, you may have put a symbol or string adjacent to a "(" when you meant it to be separated by a space or an operator. The language processor would see that as a function invocation. For example, the string `3(4+5)` should probably be written as `3*(4+5)`.

Error 44 Function did not return data

Explanation: The language processor invoked an external routine within an expression. The routine seemed to end without error, but it did not return data for use in the expression.

This may be due to specifying the name of a program that is not intended for use as a REXX function. It should be called as a command or subroutine.

Error 45 No data specified on function RETURN

Explanation: A REXX program has been called as a function, but an attempt is being made to return (by a RETURN instruction) without passing back any data. Similarly, an internal routine, called as a function, must end with a RETURN statement specifying an expression.

Error 46 Invalid variable reference

Explanation: The syntax of a variable reference is incorrect within a DROP, PARSE or PROCEDURE instruction. Check for a missing parenthesis or an incorrectly spelled variable name.

Error 48 Failure in system service

Explanation: The language processor halts execution of the program because some system service, such as stream input or output or the manipulation of the external data queue, has failed to work correctly.

Error 49 Interpretation error

Explanation: The language processor has encountered a severe error while performing a self-consistency check.

Error 115 The RXSUBCOM parameters are incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM accepts the following parameters:

- To register a subcommand environment:

```
RXSUBCOM REGISTER ENVIRONMENT_NAME DLL_NAME ENTRY_POINT
```

- To query a specific subcommand environment for existance:

```
RXSUBCOM QUERY [ENVIRONMENT_NAME [DLL_NAME]]
```

- To drop a subcommand environment handler:

```
RXSUBCOM DROP ENVIRONMENT_NAME [DLL_NAME]
```

- To load a subcommand environment from disk:

```
RXSUBCOM LOAD ENVIRONMENT_NAME [DLL_NAME]
```

Error 116 The RXSUBCOM parameter REGISTER is incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM REGISTER requires all of the following parameters:

```
RXSUBCOM REGISTER ENVIRONMENT_NAME DLL_NAME ENTRY_POINT
  ENVIRONMENT_NAME the name of the subcommand environment.
  DLL_NAME         the Dynalink Module name
  ENTRY_POINT      the name of the function to be executed when called
```

Error 117 The RXSUBCOM parameter DROP is incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM DROP requires the environment name be specified.

```
RXSUBCOM DROP ENVIRONMENT_NAME [DLL_NAME]
 ENVIRONMENT_NAME  the name of the subcommand environment
 DLL_NAME          the Dynalink Module name (optional)
```

Error 118 The RXSUBCOM parameter LOAD is incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM LOAD requires the environment name be specified.

```
RXSUBCOM LOAD ENVIRONMENT_NAME [DLL_NAME]
 ENVIRONMENT_NAME  the name of the subcommand environment
 DLL_NAME          the Dynalink Module name (optional)
```

Error 119 Invalid file name

Explanation: The name used for a file does not follow the OS/2 naming conventions. The causes may be:

- The name string contains unacceptable characters.
- The file name exceeds 8 characters.
- The file extension exceeds 3 characters.

Appendix B. Double-Byte Character Set (DBCS)

The DBCS is used to support languages that have more characters than can be represented by eight bits (such as Korean Hangeul and Japanese Kanji). REXX has a full range of DBCS functions and handling techniques. These include:

- String handling capabilities with DBCS characters.
- OPTIONS modes that handle DBCS not only as literal strings, but also in data operations.
- A number of functions that specifically support the processing of DBCS character strings.
- Defined DBCS enhancements to current instructions and functions.

Note: The use of DBCS does not effect the meaning of the built-in functions as described in Chapter 4, where we described how the characters in a result are obtained from the characters of the arguments by such actions as selecting, concatenating, and padding. This appendix describes how the resulting characters are represented as bytes. This internal representation will not normally be seen if the results are printed. It can be seen if the results are displayed on certain terminals.

General Description

The following characteristics help define the rules used by DBCS to represent the extended character set:

- Each DBCS character consists of two bytes
- There are no DBCS control characters
- The codes are within the following defined ranges, and show the valid DBCS code for the DBCS Blank.

Figure 14. DBCS Ranges

Byte	EBCDIC	ASCII
1st	X'41' to X'FE'	X'81' to X'FC'
2nd	X'41' to X'FE'	—
DBCS Blank	X'4040'	X'8140'

Note: In ASCII, the first byte may vary from country to country, but will be in the range defined previously. That is, Japan for example, only uses the ranges X'81' to X'9F' and X'E0' to X'FC' for the 1st byte.)

© Copyright IBM Corp. 1983, 1990

- DBCS alphanumeric or special symbols.

  A DBCS contains double-byte representation of alphanumeric or special symbols corresponding to those of the single-byte character set (SBCS). In EBCDIC, the first byte of a double-byte alphanumeric or special symbol is X'42' and the second is the same hexadecimal code as the corresponding EBCDIC code.

  The following are some examples:

  ```
  X'42C1' is an EBCDIC double byte A
  X'4281' is an EBCDIC double byte a
  X'427D' is an EBCDIC double byte quote
  ```

- No case translation.

  In general, there is no concept of lowercase and uppercase in DBCS.

- Notation conventions.

  Throughout this appendix, the following notational conventions will be used:

  ```
  DBCS character              ->    .A .B .C .D
  SBCS character              ->    a b c d e
  DBCS Blank                  ->    '. '
  EBCDIC Shift-out (X'0E')    ->    <
  EBCDIC Shift-in  (X'0F')    ->    >
  ```

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters are used to distinguish DBCS characters from SBCS characters.

Enabling DBCS Data Operations

The OPTIONS instruction is used to control how REXX regards DBCS data. DBCS operations are enabled using the EXMODE option. (See "OPTIONS" on page 46 for more information.)

Pure DBCS Strings and Mixed SBCS/DBCS Strings

A *pure* DBCS string consists of only DBCS characters. A mixed SBCS and DBCS string is formed by a combination of SBCS and DBCS characters. In EBCDIC, the SO and SI are used to bracket the DBCS data and distinguish it from the SBCS data. Since the SO and SI are only needed in the mixed strings, they are not associated with the pure DBCS strings.

In EBCDIC:

```
Pure DBCS string      ->      .A.B.C
Mixed string          ->      ab<.A.B>
Mixed string          ->      <.A.B>
```

In ASCII:

```
Pure DBCS string      ->      .A.B.C
Mixed string          ->      ab.A.B
```

Mixed-String Validation

The validation of mixed strings depends on the instruction, operator, or function. If an invalid mixed string is used in one that does not allow invalid mixed strings under DBCS enabled mode, it causes a SYNTAX ERROR.

The following rules must be followed for mixed-string validation:

- DBCS strings must be an even number of bytes in length.

EBCDIC only

- SO and SI must be *paired* in a string.
- Nesting of SO or SI is not permitted.

The following examples show some possible misuses:

```
'ab<cd'       ->    INVALID - not paired
'<.A<.B>.C>   ->    INVALID - nested
'<.A.BC>'     ->    INVALID - odd byte length
```

When a variable is created, modified, or referred in a REXX program under OPTIONS EXMODE, it is validated whether it contains correct mixed string or not. When a referred variable contains invalid mixed string, it depends on the instruction, function, or operator whether it causes a syntax error.

Instruction Examples

The following are some examples that illustrate how instructions work with DBCS.

PARSE

In EBCDIC:

```
      x1 = '<><.A.B><. . ><.E><.F><>'

  PARSE VAR x1 w1
            w1   ->   '<><.A.B><. . ><.E><.F><>'

  PARSE VAR x1 1 w1
            w1   ->   '<><.A.B><. . ><.E><.F><>'

  PARSE VAR x1 w1 .
            w1   ->   '<.A.B>'
The leading and trailing SO and SI are unnecessary for word parsing
and thus they are stripped off.  However, one pair is still
needed in order for a valid mixed DBCS to be returned.

  PARSE VAR x1 . w2
            w2   ->   '<. ><.E><.F><>'
Here the first blank delimited the word and the SO is added to the
string to insure the DBCS blank and the valid mixed string.
```

```
PARSE VAR x1 w1 w2
          w1   ->   '<.A.B>'
          w2   ->   '<. ><.E><.F><>'

PARSE VAR x1 w1 w2 .
          w1   ->   '<.A.B>'
          w2   ->   '<.E><.F>'
```

The word delimiting allows for unnecessary SO and SI to be dropped.

```
     x2 = 'abc<>def <.A.B><><.C.D>'

PARSE VAR x2 w1 '' w2
          w1   ->   'abc<>def <.A.B><><.C.D>'
          w2   ->   ''

PARSE VAR x2 w1 '<>' w2
          w1   ->   'abc<>def <.A.B><><.C.D>'
          w2   ->   ''

PARSE VAR x2 w1 '<><>' w2
          w1   ->   'abc<>def <.A.B><><.C.D>'
          w2   ->   ''
```

Note: For the last three examples all of the '', '< >', and '< > < >' characters are a null character, a string of length 0. When parsing, the null character matches the end of string. For this reason, w1 is assigned the value of the entire string and w2 is assigned the null string.

In ASCII:

```
      x1 = '.A.B. . .E.F'

PARSE VAR x1 w1
          w1   ->   '.A.B. . .E.F'

PARSE VAR x1 1 w1
          w1   ->   '.A.B. . .E.F'

PARSE VAR x1 w1 .
          w1   ->   '.A.B'

PARSE VAR x1 . w2
          w2   ->   '. .E.F'

PARSE VAR x1 w1 w2
          w1   ->   '.A.B'
          w2   ->   '. .E.F'

PARSE VAR x1 w1 w2 .
          w1   ->   '.A.B'
          w2   ->   '.E.F'

      x2 = 'abcdef .A.B.C.D'

PARSE VAR x2 w1 '' w2
          w1   ->   'abcdef .A.B.C.D'
          w2   ->   ''
```

PUSH and QUEUE

The PUSH and QUEUE instructions are used for adding entries to the external data queue. Since a queue entry is limited to 255 bytes, the *expression* must be truncated to less than 256 bytes. If the truncation splits a DBCS string, REXX ensures that the DBCS data integrity, that is the double byte boundary, is kept under OPTIONS EXMODE.

SAY and TRACE

When the data is split up in shorter lengths, again the DBCS data integrity is kept under OPTIONS EXMODE. In EBCDIC, if the terminal line size is less than 4, the string will be treated as SBCS data, as 4 is the minimum for mixed-string data.

DBCS Function Handling

Some built-in functions can handle DBCS. The functions that deal with word delimiting and length determining conform with the following rules under OPTIONS EXMODE:

1. **Counting characters**—Logical character lengths will be used when counting the length of a string. (That is, one byte for one SBCS logical character, while two bytes for one DBCS logical character.) In EBCDIC, SO and SI are considered to be transparent, and not counted, for every string operation.
2. **Character extraction from a string**—Characters are extracted from a string on a logical character basis. In EBCDIC, leading SO and trailing SI are not considered as part of one DBCS character. For instance, `'.A'` and `'.B'` are extracted from `'<.A.B>'`, and SO and SI are added to each DBCS character when they are finally preserved as completed DBCS characters. When multiple characters are consecutively extracted from a string, SO or SI that are between characters are also extracted. For example, `'.A><.B'` is extracted from `'<.A><.B>'`, and when the string is finally used as a completed string, the SO will prefix and the SI will suffix it to give `'<.A><.B>'`.

   In EBCDIC:

```
S1 = 'abc<>def'

SUBSTR(S1,3,1)      ->    'c'
SUBSTR(S1,4,1)      ->    'd'
SUBSTR(S1,3,2)      ->    'c<>d'

S2 = '<><.A.B><>'

SUBSTR(S2,1,1)      ->    '<.A>'
SUBSTR(S2,2,1)      ->    '<.B>'
SUBSTR(S2,1,2)      ->    '<.A.B>'
SUBSTR(S2,1,3,'x')  ->    '<.A.B><>x'
```

```
S3 = 'abc<><.A.B>'

SUBSTR(S3,3,1)      ->   'c'
SUBSTR(S3,4,1)      ->   '<.A>'
SUBSTR(S3,3,2)      ->   'c<><.A>'
DELSTR(S3,3,1)      ->   'ab<><.A.B>'
DELSTR(S3,4,1)      ->   'abc<><.B>'
DELSTR(S3,3,2)      ->   'ab<.B>'
```

In ASCII:

```
S2 = '.A.B'

SUBSTR(S2,1,1)      ->   '.A'
SUBSTR(S2,2,1)      ->   '.B'
SUBSTR(S2,1,2)      ->   '.A.B'
SUBSTR(S2,1,3,'x')  ->   '.A.Bx'

S3 = 'abc.A.B'

SUBSTR(S3,3,1)      ->   'c'
SUBSTR(S3,4,1)      ->   '.A'
SUBSTR(S3,3,2)      ->   'c.A'
DELSTR(S3,3,1)      ->   'ab.A.B'
DELSTR(S3,4,1)      ->   'abc.B'
DELSTR(S3,3,2)      ->   'ab.B'
```

3. **Character concatenation**—String concatenation can only be done with valid mixed strings. In EBCDIC, adjacent SISO or SOSI that are a result of the string concatenation are removed. Even during implicit concatenation as in the DELSTR function, unnecessary SO or SI are removed.

4. **Character comparison**—Valid mixed strings are used when comparing strings on a character basis. A DBCS character is always considered greater than a SBCS character if they are compared. In all but the strict comparisons SBCS blanks, DBCS blanks, and leading or trailing contiguous SOSI or SISO in EBCDIC are removed. SBCS blanks may be added if the lengths are not identical.

   In EBCDIC, contiguous SOSI and SISO between nonblank characters are also removed for comparison.

   Note: The strict comparison operators do not cause syntax errors even if invalid mixed strings are specified.

   In EBCDIC:

```
    '<.A>' = '<.A. >'      ->   true
'<><.A>' = '<.A><><>'      ->   true
'<>  <.A>' = '<.A>'        ->   true
'<.A><><.B>' = '<.A.B>'    ->   true
     'abc' < 'ab<. >'      ->   false
```

   In ASCII:

```
      '.A' = '.A. '        ->   true
    '. .A' = '.A'          ->   true
     'abc' < 'ab. '        ->   false
```

5. **Word extraction from a string**—*Word* means that characters in a string are delimited by a SBCS or DBCS blank.

   In EBCDIC, leading and trailing contiguous SOSI and SISO are also removed when *word*s are separated in a string, but contiguous SOSI and SISO in a word are not removed or separated for word operations. Leading and trailing contiguous SOSI and SISO of a word are not removed if they are among words that are extracted at the same time.

   In EBCDIC:

```
W1 = '<><. .A. . .B><.C. .D><>'

SUBWORD(W1,1,1)      ->    '<.A>'
SUBWORD(W1,1,2)      ->    '<.A. . .B><.C>'
SUBWORD(W1,3,1)      ->    '<.D>'
SUBWORD(W1,3)        ->    '<.D>'

W2 = '<.A. .B><.C><> <.D>'

SUBWORD(W2,2,1)      ->    '<.B><.C>'
SUBWORD(W2,2,2)      ->    '<.B><.C><> <.D>'
```

   In ASCII:

```
W1 = '. .A. . .B.C. .D'

SUBWORD(W1,1,1)      ->    '.A'
SUBWORD(W1,1,2)      ->    '.A. . .B.C'
SUBWORD(W1,3,1)      ->    '.D'
SUBWORD(W1,3)        ->    '.D'

W2 = '.A. .B.C .D'

SUBWORD(W2,2,1)      ->    '.B.C'
SUBWORD(W2,2,2)      ->    '.B.C .D'
```

Built-In Function Examples

Examples for built-in functions, those that support DBCS and follow the rules defined, are given in this section. For full function descriptions and the syntax diagrams, refer to Chapter 4, "Functions."

ABBREV

The following examples show the rules for applying the character comparison and character extraction from a string.

In EBCDIC:

```
ABBREV('<.A.B.C>','<.A.B>')     ->    1
ABBREV('<.A.B.C>','<.A.C>')     ->    0
ABBREV('<.A><.B.C>','<.A.B>')   ->    1
ABBREV('aa<>bbccdd','aabbcc')   ->    1
```

In ASCII:

```
ABBREV('.A.B.C','.A.B')         ->    1
ABBREV('.A.B.C','.A.C')         ->    0
```

COMPARE

The following examples show the rules for applying the character concatenation for padding, the character extraction from a string, and character comparison.

In EBCDIC:

```
COMPARE('<.A.B.C>','<.A.B><.C>')      ->  0
COMPARE('<.A.B.C>','<.A.B.D>')        ->  3
COMPARE('ab<>cde','abcdx')            ->  5
COMPARE('<.A><>','<.A>','<. >')       ->  0
```

In ASCII:

```
COMPARE('.A.B.C','.A.B.C')            ->  0
COMPARE('.A.B.C','.A.B.D')            ->  3
COMPARE('abcde','abcdx')              ->  5
```

COPIES

The following examples show the rules for applying the character concatenation.

In EBCDIC:

```
COPIES('<.A.B>',2)        ->  '<.A.B.A.B>'
COPIES('<.A><.B>',2)      ->  '<.A><.B.A><.B>'
COPIES('<.A.B><>',2)      ->  '<.A.B><.A.B><>'
```

In ASCII:

```
COPIES('.A.B',2)          ->  '.A.B.A.B'
```

DATATYPE

The following examples show the rules for applying the character extraction from a string and the character comparison rules.

```
DATATYPE('<.A.B>')        ->  'CHAR'
DATATYPE('<.A.B>','D')    ->  1
DATATYPE('<.A.B>','C')    ->  1
DATATYPE('a<.A.B>b','D')  ->  0
DATATYPE('a<.A.B>b','C')  ->  1
DATATYPE('abcde','C')     ->  0
DATATYPE('<.A.B','C')     ->  0
```

INSERT and OVERLAY

The following examples show the rules for applying the character extraction from a string and character comparison.

In EBCDIC:

```
INSERT('a','b<><.A.B>',1)              -> 'ba<><.A.B>'
INSERT('<.A.B>','<.C.D><>',2)          -> '<.C.D.A.B><>'
INSERT('<.A.B>','<.C.D><><.E>',2)      -> '<.C.D.A.B><><.E>'
INSERT('<.A.B>','<.C.D><>',3,,'<.E>') -> '<.C.D><.E.A.B>'

OVERLAY('<.A.B>','<.C.D><>',2)           -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',2)       -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',3)       -> '<.C.D><><.A.B>'
OVERLAY('<.A.B>','<.C.D><>',4,,'<.E>')   -> '<.C.D><.E.A.B>'
OVERLAY('<.A>','<.C.D><.E>',2)           -> '<.C.A><.E>'
```

In ASCII:

```
INSERT('a','b.A.B',1)              -> 'ba.A.B'
INSERT('.A.B','.C.D',2)            -> '.C.D.A.B'
INSERT('.A.B','.C.D.E',2)          -> '.C.D.A.B.E'
INSERT('.A.B','.C.D',3,,'.E')      -> '.C.D.E.A.B'

OVERLAY('.A.B','.C.D',2)           -> '.C.A.B'
OVERLAY('.A.B','.C.D.E',2)         -> '.C.A.B'
OVERLAY('.A.B','.C.D.E',3)         -> '.C.D.A.B'
OVERLAY('.A.B','.C.D',4,,'.E')     -> '.C.D.E.A.B'
OVERLAY('.A','.C.D.E',2)           -> '.C.A.E'
```

LEFT, RIGHT, and CENTER

The following examples show the rules for applying the character concatenation for padding and character extraction from a string.

In EBCDIC:

```
LEFT('<.A.B.C.D.E>',4)      ->   '<.A.B.C.D>'
LEFT('a<>',2)               ->   'a<> '
LEFT('<.A>',2,'*')          ->   '<.A>*'
RIGHT('<.A.B.C.D.E>',4)     ->   '<.B.C.D.E>'
RIGHT('a<>',2)              ->   ' a'
CENTER('<.A.B>',10,'<.E>') ->   '<.E.E.E.E.A.B.E.E.E.E>'
CENTER('<.A.B>',11,'<.E>') ->   '<.E.E.E.E.A.B.E.E.E.E.E>'
CENTER('<.A.B>',10,'e')     ->   'eeee<.A.B>eeee'
```

In ASCII:

```
LEFT('.A.B.C.D.E',4)        ->   '.A.B.C.D'
LEFT('a',2)                 ->   'a '
LEFT('.A',2,'*')            ->   '.A*'
RIGHT('.A.B.C.D.E',4)       ->   '.B.C.D.E'
RIGHT('a',2)                ->   ' a'
CENTER('.A.B',10,'.E')      ->   '.E.E.E.E.A.B.E.E.E.E'
CENTER('.A.B',11,'.E')      ->   '.E.E.E.E.A.B.E.E.E.E.E'
CENTER('.A.B',10,'e')       ->   'eeee.A.Beeee'
```

LENGTH

The following examples show the rules for applying the counting characters.

In EBCDIC:

```
LENGTH('<.A.B><.C.D><>')      ->   4
```

In ASCII:

```
LENGTH('.A.B.C.D')            ->   4
```

LINEIN

When reading a line from a stream under OPTIONS EXMODE, the LINEIN function ignores any DBCS strings when searching for line-end characters. The DBCS string itself is read normally from the stream.

REVERSE

The following examples show the rules for applying the character extraction from a string and character concatenation.

In EBCDIC:

```
REVERSE('<.A.B><.C.D><>')      ->   '<><.D.C><.B.A>'
```

In ASCII:

```
REVERSE('.A.B.C.D')            ->   '.D.C.B.A'
```

SPACE

The following examples show the rules for applying the word extraction from a string and character concatenation.

In EBCDIC:

```
SPACE('a<.A.B. .C.D>',1)         ->  'a<.A.B> <.C.D>'
SPACE('a<.A><>. .C.D>',1,'x')    ->  'a<.A>x<.C.D>'
SPACE('a<.A><. .C.D>',1,'<.E>') ->  'a<.A.E.C.D>'
```

In ASCII:

```
SPACE('a.A.B. .C.D',1)           ->  'a.A.B .C.D'
SPACE('a.A. .C.D',1,'x')         ->  'a.Ax.C.D'
SPACE('a.A. .C.D',1,'.E')        ->  'a.A.E.C.D'
```

STRIP

The following examples show the rules for applying the character extraction from a string and character concatenation.

In EBCDIC:

```
STRIP('<><.A><.B><.A><>',,'<.A>')   ->  '<.B>'
```

In ASCII:

```
STRIP('.A.B.A',,'.A')               ->  '.B'
```

SUBSTR and DELSTR

The following examples show the rules for applying the character extraction from a string and character concatenation.

In EBCDIC:

```
SUBSTR('<><.A><><.B><.C.D>',1,2)  ->  '<.A><><.B>'
DELSTR('<><.A><><.B><.C.D>',1,2)  ->  '<><.C.D>'
SUBSTR('<.A><><.B><.C.D>',2,2)    ->  '<.B><.C>'
DELSTR('<.A><><.B><.C.D>',2,2)    ->  '<.A><><.D>'
SUBSTR('<.A.B><>',1,2)            ->  '<.A.B>'
SUBSTR('<.A.B><>',1)              ->  '<.A.B><>'
```

In ASCII:

```
SUBSTR('.A.B.C.D',1,2)            ->  '.A.B'
DELSTR('.A.B.C.D',1,2)            ->  '.C.D'
SUBSTR('.A.B.C.D',2,2)            ->  '.B.C'
DELSTR('.A.B.C.D',2,2)            ->  '.A.D'
SUBSTR('.A.B',1,2)                ->  '.A.B'
SUBSTR('.A.B',1)                  ->  '.A.B'
```

SUBWORD and DELWORD

The following examples show the rules for applying the word extraction from a string and character concatenation.

In EBCDIC:

```
SUBWORD('<><. .A. . .B><.C. .D>',1,2) ->  '<.A. . .B><.C>'
DELWORD('<><. .A. . .B><.C. .D>',1,2) ->  '<><. .D>'
SUBWORD('<><.A. . .B><.C. .D>',1,2)   ->  '<.A. . .B><.C>'
DELWORD('<><.A. . .B><.C. .D>',1,2)   ->  '<><.D>'
SUBWORD('<.A. .B><.C><> <.D>',1,2)    ->  '<.A. .B><.C>'
DELWORD('<.A. .B><.C><> <.D>',1,2)    ->  '<.D>'
```

In ASCII:

```
SUBWORD('. .A. . .B.C. .D',1,2)       ->  '.A. . .B.C'
DELWORD('. .A. . .B.C. .D',1,2)       ->  '. .D'
SUBWORD('.A. . .B.C. .D',1,2)         ->  '.A. . .B.C'
DELWORD('.A. . .B.C. .D',1,2)         ->  '.D'
SUBWORD('.A. .B.C .D',1,2)            ->  '.A. .B.C'
DELWORD('.A. .B.C .D',1,2)            ->  '.D'
```

TRANSLATE

The following examples show the rules for applying the character extraction from a string and character concatenation.

In EBCDIC:

```
TRANSLATE('abcd','<.A.B.C>','abc')        ->  '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','abc')      ->  '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','ab<>c')    ->  '<.A.B.C>d'
TRANSLATE('a<>bcd','<><.A.B.C>','ab<>c')  ->  '<.A.B.C>d'
TRANSLATE('a<>xcd','<><.A.B.C>','ab<>c')  ->  '<.A>x<.C>d'
```

In ASCII:

```
TRANSLATE('abcd','.A.B.C','abc')          ->  '.A.B.Cd'
TRANSLATE('axcd','.A.B.C','abc')          ->  '.Ax.Cd'
```

VERIFY

The following examples show the rules for applying the character extraction from a string, character comparison, and character concatenation.

In EBCDIC:

```
VERIFY('<><.A.B><.X>','<.B.A.C.D.E>')   ->  3
```

In ASCII:

```
VERIFY('.A.B.X','.B.A.C.D.E')           ->  3
```

WORD,WORDINDEX, and WORDLENGTH

The following examples show the rules for applying the word extraction from a string and counting characters (for WORDINDEX and WORDLENGTH).

In EBCDIC:

```
    X = '<><. .A. . .B><.C. .D>'

  WORD(X,1)           ->    '<.A>'
  WORDINDEX(X,1)      ->    2
  WORDLENGTH(X,1)     ->    1

    Y = '<><.A. . .B><.C. .D>'

  WORD(Y,1)           ->    '<.A>'
  WORDINDEX(Y,1)      ->    1
  WORDLENGTH(Y,1)     ->    1

    Z = '<AA  BB><CC> <DD>'

  WORD(Z,2)           ->    '<.B><.C>'
  WORDINDEX(Z,2)      ->    3
  WORDLENGTH(Z,2)     ->    2
```

In ASCII:

```
    X = '. .A. . .B.C. .D'

  WORD(X,1)           ->    '.A'
  WORDINDEX(X,1)      ->    2
  WORDLENGTH(X,1)     ->    1

    Y = '.A. . .B.C. .D'

  WORD(Y,1)           ->    '.A'
  WORDINDEX(Y,1)      ->    1
  WORDLENGTH(Y,1)     ->    1

    Z = '.A. .B.C .D'
                    Where .A is followed by a DBCS blank
                    and .C is followed by an SBCS blank

  WORD(Z,2)           ->    '.B.C'
  WORDINDEX(Z,2)      ->    3
  WORDLENGTH(Z,2)     ->    2
```

WORDS

The following examples show the rules for applying the word extraction from a string.

In EBCDIC:

```
    X = '<><. .A. . .B><.C. .D>'

  WORDS(X)            ->    3
```

In ASCII:

```
X = '. .A. . .B.C. .D'

WORDS(X)            ->    3
```

WORDPOS

The following examples show the rules for applying the word extraction from a string and character comparison.

In EBCDIC:

```
WORDPOS('<.B.C> abc','<.A. .B.C> abc')              ->   2
WORDPOS('<.A.B>','<.A.B. .A.B><. .B.C. .A.B>',3)    ->   4
```

In ASCII:

```
WORDPOS('.B.C abc','.A. .B.C abc')                  ->   2
WORDPOS('.A.B','.A.B. .A.B. .B.C. .A.B',3)          ->   4
```

DBCS Processing Functions

The following text describes the functions that support DBCS mixed string. These functions handle mixed strings regardless of the OPTIONS *mode*.

Note: When used with DBCS functions, *length* is always measured in bytes (as opposed to LENGTH(string), which is measured in characters).

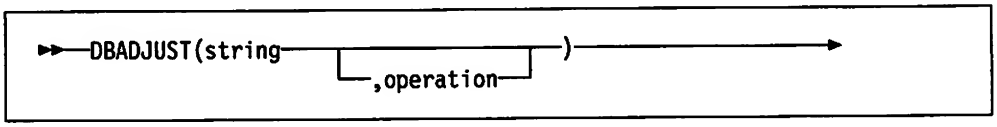
Counting Option

In EBCDIC, when specified in the functions, the counting option can be used to control whether or not the SO and SI are considered present when determining the length. If `Y` is specified, SO and SI within mixed strings are counted. `N` specifies not to count the SO and SI, and is the default.

In ASCII, the count option is ignored if `Y` is specified and the default `N` is always in effect. As such, it will yield different results between EBCDIC and ASCII implementations.

Function Descriptions

DBADJUST



DBADJUST, in ASCII, returns the input string; in EBCDIC, adjusts all contiguous SI through SO and SO through SI characters in *string* based on the *operation* specified. Valid operations (of which only the capitalized letter is significant, all others are ignored) are:

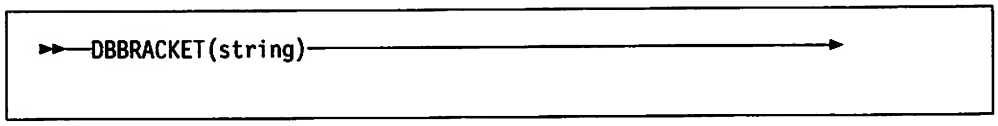
Blank Changes contiguous characters to blanks (X'4040').

Remove Removes contiguous characters, and is the default.

The following are some EBCDIC examples:

```
DBADJUST('<.A><.B>a<>b','B')    ->    '<.A. .B>a  b'
DBADJUST('<.A><.B>a<>b','R')    ->    '<.A.B>ab'
DBADJUST('<><.A.B>','B')        ->    '<. .A.B>'
```

DBBRACKET

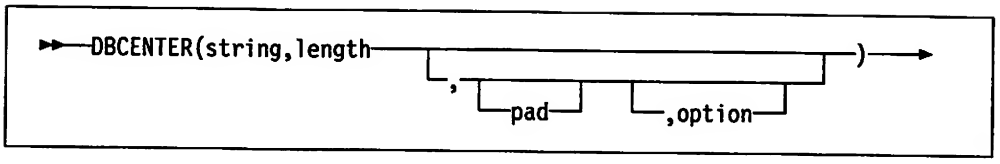


DBBRACKET, in ASCII, returns the input string; in EBCDIC, adds SO through SI brackets to a pure DBCS string. If *string* is not a pure DBCS string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

The following are some EBCDIC examples:

```
DBBRACKET('.A.B')       ->    '<.A.B>'
DBBRACKET('abc')        ->    SYNTAX error
DBBRACKET('<.A.B>')     ->    SYNTAX error
```

DBCENTER



DBCENTER returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up the extra length. The default *pad* character is a blank. If the string is longer than *length*, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right end loses or gains one more character than the left end.

Option is used to control the counting rule. `Y` will count SO and SI within mixed strings as one each. `N` will not count the SO and SI and is the default.

Note: In ASCII, *option* `Y` is ignored and `N` is always in effect.

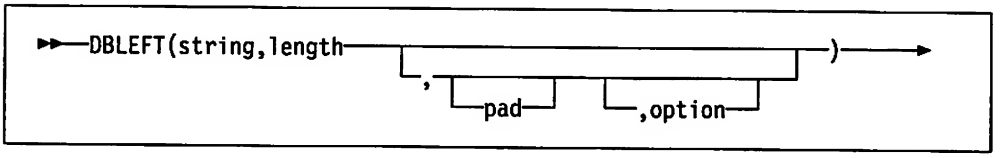
The following are some EBCDIC examples:

```
DBCENTER('<.A.B.C>',4)              ->    ' <.B> '
DBCENTER('<.A.B.C>',3)              ->    ' <.B>'
DBCENTER('<.A.B.C>',10,'x')         ->    'xx<.A.B.C>xx'
DBCENTER('<.A.B.C>',10,'x','Y')     ->    'x<.A.B.C>x'
DBCENTER('<.A.B.C>',4,'x','Y')      ->    '<.B>'
DBCENTER('<.A.B.C>',5,'x','Y')      ->    'x<.B>'
DBCENTER('<.A.B.C>',8,'<.P>')       ->    ' <.A.B.C> '
DBCENTER('<.A.B.C>',9,'<.P>')       ->    ' <.A.B.C.P>'
DBCENTER('<.A.B.C>',10,'<.P>')      ->    '<.P.A.B.C.P>'
DBCENTER('<.A.B.C>',12,'<.P>','Y') ->    '<.P.A.B.C.P>'
```

The following are some ASCII examples:

```
DBCENTER('.A.B.C',9,'.P')      ->    ' .A.B.C.P'
DBCENTER('.A.B.C',10,'.P')     ->    '.P.A.B.C.P'
```

DBLEFT



DBLEFT returns a string of length *length* containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank.

Option is used to control the counting rule. `Y` will count SO and SI within mixed strings as one each. `N` will not count the SO and SI and is the default.

Note: In ASCII, *option* `Y` is ignored and `N` is always in effect.

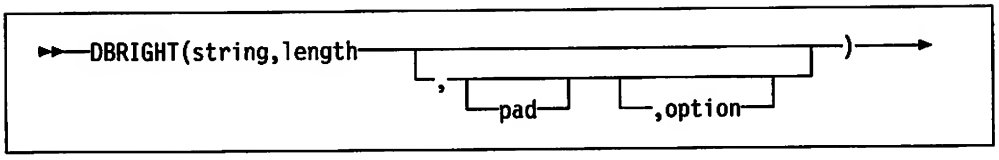
The following are some EBCDIC examples:

```
DBLEFT('ab<.A.B>',4)              ->    'ab<.A>'
DBLEFT('ab<.A.B>',3)              ->    'ab '
DBLEFT('ab<.A.B>',4,'x','Y')      ->    'abxx'
DBLEFT('ab<.A.B>',3,'x','Y')      ->    'abx'
DBLEFT('ab<.A.B>',8,'<.P>')       ->    'ab<.A.B.P>'
DBLEFT('ab<.A.B>',9,'<.P>')       ->    'ab<.A.B.P> '
DBLEFT('ab<.A.B>',8,'<.P>','Y')   ->    'ab<.A.B>'
DBLEFT('ab<.A.B>',9,'<.P>','Y')   ->    'ab<.A.B> '
```

The following are some ASCII examples:

```
DBLEFT('ab.A.B',3,,'Y')           ->    'ab '
DBLEFT('ab.A.B',9,'.P')           ->    'ab.A.B.P '
```

DBRIGHT



DBRIGHT returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank.

Option is used to control the counting rule. `Y` will count SO and SI within mixed strings as one each. `N` will not count the SO and SI and is the default.

Note: In ASCII, *option* `Y` is ignored and `N` is always in effect.

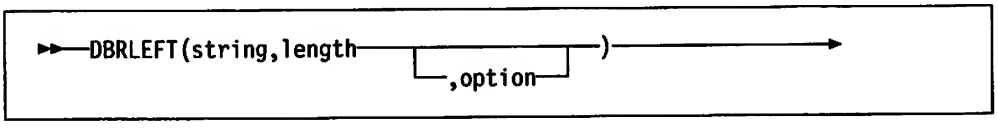
The following are some EBCDIC examples:

```
DBRIGHT('ab<.A.B>',4)             ->    '<.A.B>'
DBRIGHT('ab<.A.B>',3)             ->    ' <.B>'
DBRIGHT('ab<.A.B>',5,'x','Y')     ->    'x<.B>'
DBRIGHT('ab<.A.B>',10,'x','Y')    ->    'xxab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>')      ->    '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',9,'<.P>')      ->    ' <.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>','Y')  ->    'ab<.A.B>'
DBRIGHT('ab<.A.B>',11,'<.P>','Y') ->    '   ab<.A.B>'
DBRIGHT('ab<.A.B>',12,'<.P>','Y') ->    '<.P>ab<.A.B>'
```

The following are some ASCII examples:

```
DBRIGHT('ab.A.B',3)               ->    ' .B'
DBRIGHT('ab.A.B',12,'.P','Y')     ->    '.P.P.Pab.A.B'
```

DBRLEFT



DBRLEFT returns the remainder from the DBLEFT function of *string*. If *length* is greater than the length of *string*, a null string is returned.

Option is used to control the counting rule. Y will count SO and SI within mixed strings as one each. N will not count the SO and SI and is the default.

Note: In ASCII, *option* Y is ignored and N is always in effect.

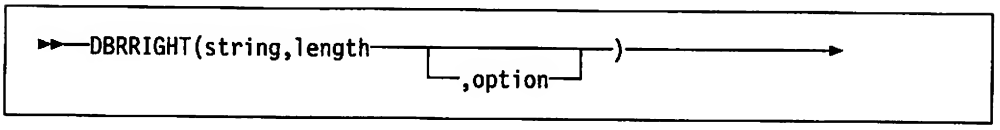
The following are some EBCDIC examples:

```
DBRLEFT('ab<.A.B>',4)          ->   '<.B>'
DBRLEFT('ab<.A.B>',3)          ->   '<.A.B>'
DBRLEFT('ab<.A.B>',4,'Y')      ->   '<.A.B>'
DBRLEFT('ab<.A.B>',3,'Y')      ->   '<.A.B>'
DBRLEFT('ab<.A.B>',8)          ->   ''
DBRLEFT('ab<.A.B>',9,'Y')      ->   ''
```

The following are some ASCII examples:

```
DBRLEFT('ab.A.B',3)            ->   '.A.B'
DBRLEFT('ab.A.B',9,'Y')        ->   ''
```

DBRRIGHT



DBRRIGHT returns the remainder from the DBRIGHT function of *string*. If *length* is greater than the length of *string*, a null string is returned.

Option is used to control the counting rule. Y will count SO and SI within mixed strings as one each. N will not count the SO and SI and is the default.

Note: In ASCII, *option* Y is ignored and N is always in effect.

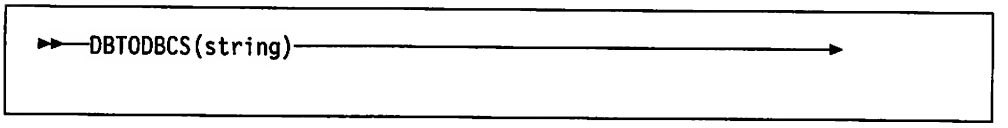
The following are some EBCDIC examples:

```
DBRRIGHT('ab<.A.B>',4)         ->   'ab'
DBRRIGHT('ab<.A.B>',3)         ->   'ab<.A>'
DBRRIGHT('ab<.A.B>',5)         ->   'a'
DBRRIGHT('ab<.A.B>',4,'Y')     ->   'ab<.A>'
DBRRIGHT('ab<.A.B>',5,'Y')     ->   'ab<.A>'
DBRRIGHT('ab<.A.B>',8)         ->   ''
DBRRIGHT('ab<.A.B>',8,'Y')     ->   ''
```

The following are some ASCII examples:

```
DBRRIGHT('ab.A.B',3)           ->   'ab.A'
DBRRIGHT('ab.A.B',8)           ->   ''
```

DBTODBCS



DBTODBCS converts all passed, valid SBCS characters (including the SBCS blank) within *string* to the corresponding DBCS equivalents. Other single-byte codes and all

DBCS characters are not changed. In EBCDIC, SO and SI brackets are added and removed where appropriate.

The following are some EBCDIC examples:

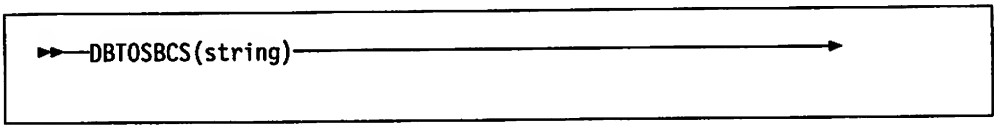
```
DBTODBCS('Rexx 1988')       ->    '<.R.e.x.x. .1.9.8.8>'
DBTODBCS('<.A> <.B>')       ->    '<.A. .B>'
```

The following are some ASCII examples:

```
DBTODBCS('Rexx 1988')       ->    '.R.e.x.x. .1.9.8.8'
DBTODBCS('.A .B')           ->    '.A. .B'
```

Note: In the above examples, the .x is the DBCS character corresponding to a SBCS x.

DBTOSBCS



DBTOSBCS converts all passed, valid DBCS characters (including the DBCS blank) within string to the corresponding SBCS equivalents. Other DBCS characters and all SBCS characters are not changed. In EBCDIC, SO and SI brackets are removed where appropriate.

The following are some EBCDIC examples:

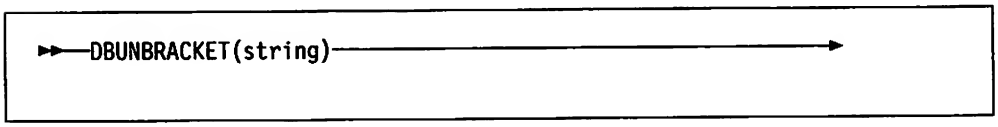
```
DBTOSBCS('<.S.d>/<.2.-.1>')     ->    'Sd/2-1'
DBTOSBCS('<.X. .Y>')            ->    '<.X> <.Y>'
```

The following are some ASCII examples:

```
DBTOSBCS('.S.d/.2.-.1')         ->    'Sd/2-1'
DBTOSBCS('.X. .Y')              ->    '.X .Y'
```

Note: In the above examples, the .d is the DBCS character corresponding to a SBCS d. The .X and .Y do not have an SBCS corresponding character, and are not converted.

DBUNBRACKET

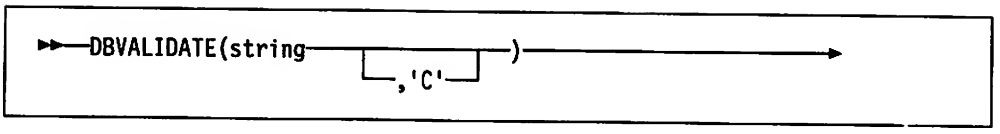


DBUNBRACKET, in ASCII, returns the input string; in EBCDIC, removes the SO through SI brackets from a pure DBCS *string* enclosed by SO and SI brackets. If the *string* is not bracketed, a SYNTAX error results.

The following are some EBCDIC examples:

```
DBUNBRACKET('<.A.B>')       ->    '.A.B'
DBUNBRACKET('ab<.A>')       ->    SYNTAX error
```

DBVALIDATE



DBVALIDATE returns 1 if the *string* is a valid mixed string or SBCS string. Otherwise, 0 is returned. Mixed-string validation rules are:

1. Only valid DBCS character codes
2. DBCS string is an even number of bytes in length
3. EBCDIC only—Proper SO through SI pairing.

In ASCII, the option C has no effect. In EBCDIC, if C is omitted, only the leftmost byte of each DBCS character is checked to see that it falls in the valid range for the implementation it is being run on. (That is, in EBCDIC, the leftmost byte range is from X'41' to X'FE'.)

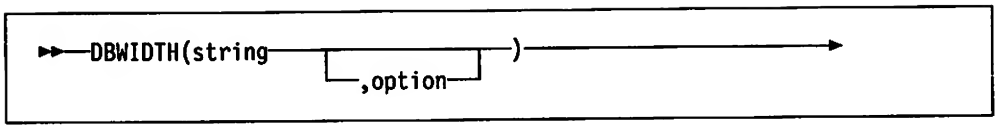
The following are some EBCDIC examples:

```
   x='abc<de'

DBVALIDATE('ab<.A.B>')          ->   1
DBVALIDATE(x)                   ->   0

   y='C1C20E111213140F'X

DBVALIDATE(y)                   ->   1
DBVALIDATE(y,'C')               ->   0
```

The following are some ASCII examples:

```
DBVALIDATE('ab.A.B')  ->   1
DBVALIDATE('ab.A.')   ->   0  /* widow left DBCS byte */

   y='C1C2FCFFFCFF'X      /* 'FCFF'x will pass DBCS range check */

DBVALIDATE(y)                  ->   1
DBVALIDATE(y,'C')              ->   1
```

DBWIDTH



DBWITH returns the length of *string* in bytes.

Option is used to control the counting rule. `Y` will count SO and SI within mixed strings as one each. `N` will not count the SO and SI and is the default.

Note: In ASCII, *option* `Y` is ignored and `N` is always in effect.

The following are some EBCDIC examples:

```
DBWIDTH('ab<.A.B>','Y')    ->    8
DBWIDTH('ab<.A.B>','N')    ->    6
```

The following are some ASCII examples:

```
DBWIDTH('ab.A.B','Y')      ->    6
DBWIDTH('ab.A.B','N')      ->    6
```

Index

A

B

© Copyright IBM Corp. 1983, 1990

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Special Characters

™ Operating System/2 is a trademark of
International Business Machines Corporation.
® IBM is a registered trademark of
International Business Machines Corporation.

©IBM Corp.1989, 1990
Printed in the
United States of America
All Rights Reserved

64F3057

Order No. 01F0271
S01F-0271

S01F-0271-00